IMPLEMENTATION OF A VOICE ACTIVITY DETECTION AND COMFORT NOISE GENERATION ALGORITHM

JING LIANG

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements for the Degree
of Master of Applied Science at Concordia University
Montreal, Quebec, Canada

April 2004

Your file Votre référence
ISBN: 0-612-91066-0
Our file Notre référence
ISBN: 0-612-91066-0

ABSTRACT

IMPLEMENTATION OF A VOICE ACTIVITY DETECTION AND COMFORT NOISE GENERATION ALGORITHM

Jing Liang

Voice activity detection and comfort noise generation (VAD-CNG) algorithms are widely employed in packet voice communication systems to reduce transmission bandwidth. Full-duplex voice communication systems are typically interactive, requiring a very short end-to-end delay. Therefore, an effective implementation is critical to make any VAD-CNG algorithm a practical solution in a real-time embedded system.

In practice, algorithms are generally implemented on programmable digital signal processors (DSPs) due to the availability of such processors at a very low cost. The family of digital signal processors TMS320C54xx from Texas Instruments (TI) has been the most widely used one in wireless communications, voice over IP gateway, consumer electronics and broadband communications.

This thesis is devoted to the investigation of effective implementations of a modified version of a well-established fixed-point data-dependent VAD-CNG algorithm of Nortel Networks. In this study, the algorithm to be implemented is first described in detail. Then the target platform, a TMS320C5402DSK DSP board, along with its real-time operating system (DSP/BIOS) and its associated analysis tools, is introduced. Two implementation schemes are presented in this thesis. The first one is a direct implementation of the modified algorithm on the DSP board. The second one is an implementation, wherein some optimizations that target the reduction of the implementational complexity of the algorithm

are introduced. Several DSP/BIOS real-time analysis tools provided by the DSP board and some speech samples are used to test the performance of the implementations. Experimental results are presented to show the effectiveness of the optimizations for the modified VAD-CNG algorithm. These results show that over 80% of the reduction in the implementational complexity is achieved through the proposed optimizations, making it possible to incorporate such a VAD-CNG algorithm into a practical real-time voice communication system. A real-time audio codec system is built in the laboratory to demonstrate the real-time implementation of this algorithm. This demonstration system should also serve as a useful experimental tool for further investigation on this topic.

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my thesis supervisors, Dr. M. N. S. Swamy and Dr. M. O. Ahmad, for their guidance and support throughout the course of this research work.

I am greatly thankful to Dr. Jimmy Zhang from Texas Instruments Inc., who has spent a lot of time to review my thesis and give me many precious suggestions.

I am grateful to the colleagues of our research group, particularly, Mr. Wei Chu, Mr. Jianguang Zhou and Dr. Ke Li, for their continuous support, suggestions and inspirations during this research.

I would like to thank Nortel Networks and Micronet, a National Network of Centers of Excellence, Canada, for their financial support to this program.

TABLE OF CONTENTS

LIST OF FIGURES ..... **x**

LIST OF TABLES ..... **xii**

LIST OF ABBREVIATIONS ..... **xiii**

CHAPTER 1

INTRODUCTION **1**

1.1 PRINCIPLE OF VAD-CNG ALGORITHMS 1

1.2 NORTEL VAD-CNG ALGORITHM 3

1.2.1 VAD Algorithm *3*

1.2.1.1 LPC Analysis 4

1.2.1.2 Prediction Gain and Non-stationarity 7

1.2.1.3 Energy Analysis 8

1.2.1.4 Likelihood 9

1.2.2 CNG Algorithm *10*

1.2.2.1 CNG Encoder 11

1.2.2.2 CNG Decoder 12

1.2.3 DTX Algorithm *14*

1.3 MODIFIED NORTEL VAD-CNG ALGORITHM 15

1.3.1 Revisions to Decision Rules *15*

1.3.2 Use of a Rectangular Window *16*

1.3.3 Simplification of the Highpass Filter *16*

1.3.4 Reduction in Frequency of SID Packets Transmission *17*

1.3.5 Other Modifications *18*

1.4 MOTIVATION AND SCOPE OF THE THESIS 18

1.5 ORGANIZATION OF THE THESIS 19

CHAPTER 2

AN OVERVIEW OF TMS320C5402 DSP KIT 20

2.1 HARDWARE OVERVIEW 20

2.1.1 TMS320VC5402 DSP *21*

2.1.2 CY37128 CPLD *22*

2.1.3 External Memory *22*

2.1.4 Microphone/Speaker Interface *23*

2.2 SOFTWARE OVERVIEW 23

2.2.1 Code Composer Studio *23*

2.2.2 DSP/BIOS *25*

2.2.2.1 Configuration Tool 26

2.2.2.2 Thread Scheduling 27

2.2.2.3 Memory Management 29

2.2.2.4 Input and Output Management 30

2.3 REAL-TIME ANALYSIS TOOLS 32

2.3.1 Execution Graph *32*

2.3.2 CPU Load Graph *33*

2.3.3 Statistics View *34*

2.4 CONCLUSION 35

CHAPTER 3

DIRECT IMPLEMENTATION OF THE MODIFIED ALGORITHM 36

3.1 C MODULES OF VAD ALGORITHM 36

3.2 TEST VECTORS AND COMPLEXITY 39

3.2.1 Test Vectors 40

3.2.2 Complexity and Average Complexity 40

3.3 IMPLEMENTATION USING AN ON-BOARD TEST VECTOR 41

3.3.1 Implementation Process 41

3.3.2 Limitation of This Implementation 43

3.4 IMPLEMENTATION USING HOST CHANNELS 44

3.5 PERFORMANCE ANALYSIS 47

3.6 CONCLUSION 50

CHAPTER 4

OPTIMIZED IMPLEMENTATION OF THE MODIFIED ALGORITHM..... 52

4.1 INTRODUCTION 52

4.2 OPTIMIZATIONS OF THE DIRECT IMPLEMENTATION 53

4.2.1 Performing Program-level Optimization 53

4.2.2 Rewriting Some C Routines in Assembly Language 57

4.2.3 Using Intrinsic Functions 60

4.2.4 Using Assembly-Optimized Functions in DSPLIB 61

4.2.4.1 Cascaded IIR Direct Form II Using 4-Coefficients per Biquad 62

4.2.4.2 FIR Filter 64

4.2.5 Reorganizing the Memory 65

4.3 EXPERIMENTAL RESULTS 69

4.4 SUMMARY 72

CHAPTER 5

A REAL-TIME DEMONSTRATION SYSTEM 73

5.1 OVERVIEW OF THE REAL-TIME SYSTEM 73

5.2 HARDWARE INTERFACES AND DATAFLOW 74

5.3 DSP SCHEDULING 76

5.3.1 HWI in the Application 76

5.3.2 SWIs in the Application 77

5.4 EXPERIMENTAL RESULTS 80

5.4.1 Execution Graph 80

5.4.2 CPU Load Graph 81

CHAPTER 6

CONCLUSION AND FUTURE RESEARCH DIRECTIONS 84

6.1 CONCLUSION 84

6.2 FUTURE RESEARCH DIRECTIONS 85

APPENDIX 86

APPENDIX A. SOURCE CODE OF DIRECT IMPLEMENTATION 86

APPENDIX B. SOURCE CODE OF OPTIMIZED IMPLEMENTATION 89

APPENDIX C. SOURCE CODE OF REAL-TIME IMPLEMENTATION 89

REFERENCES 91

LIST OF FIGURES

FIGURE 1.1 VAD/DTX/CNG BLOCK DIAGRAM [4] 2

FIGURE 1.2 VAD BLOCK DIAGRAM [18] 4

FIGURE 1.3 COMFORT NOISE GENERATION SCHEME [18] 13

FIGURE 2.1 BLOCK DIAGRAM OF CCS [26] 24

FIGURE 2.2 CONFIGURATION TOOL 26

FIGURE 2.3 PIP MODULE 30

FIGURE 2.4 HOST CHANNEL CONTROL 31

FIGURE 2.5 EXECUTION GRAPH 33

FIGURE 2.6 CPU LOAD GRAPH 33

FIGURE 2.7 STATISTICS VIEW 34

FIGURE 3.1 FLOWCHART OF THE *VAD* MODULE 38

FIGURE 3.2 STATISTIC VIEW OF THE IMPLEMENTATION USING AN ON-BOARD TEST VECTOR 43

FIGURE 3.3 STATISTIC VIEW OF THE IMPLEMENTATION USING ANOTHER ON-BOARD TEST VECTOR 44

FIGURE 3.4 BLOCK DIAGRAM OF IMPLEMENTATION USING HOST CHANNELS 45

FIGURE 3.5 STATISTIC VIEW OF THE IMPLEMENTATION OF THE MODIFIED ALGORITHM 49

FIGURE 4.1 SETTING THE COMPLIER 56

FIGURE 4.2 MEMORY MAPS (A) BEFORE REORGANIZING MEMORY (B) AFTER REORGANIZING MEMORY 67

FIGURE 5.1 BLOCK DIAGRAM OF THE REAL-TIME SYSTEM 74

FIGURE 5.2 HARDWARE CONNECTION AND DATAFLOW OF THE REAL-TIME SYSTEM 75

FIGURE 5.3 DEFINING HWI_SINT10 77

FIGURE 5.4 DEFINING INPUT_PIP 77

FIGURE 5.5 DEFINING OUTPUT_PIP 78

FIGURE 5.6 DSS_RXPRIME AND DSS_RXPRIME 79

FIGURE 5.7 EXECUTION GRAPHS OF THE REAL-TIME SYSTEM BASED ON (A) DIRECT IMPLEMENTATION AND (B) OPTIMIZED IMPLEMENTATION 81

FIGURE 5.8 CPU LOAD GRAPHS OF THE REAL-TIME SYSTEMBASED ON (A) DIRECT IMPLEMENTATION AND (B) OPTIMIZED IMPLEMENTATION 82

FIGURE 5.9 CPU LOAD GRAPHS OF THE REAL-TIME SYSTEM WITH DIFFERENT VOLUME BASED ON (A) DIRECT IMPLEMENTATION AND (B) OPTIMIZED IMPLEMENTATION 83

LIST OF TABLES

TABLE 2.1 NAMES OF MEMORY SEGMENTS 29

TABLE 3.1 COMPLEXITIES AND AVERAGE COMPLEXITIES OF THE ORIGINAL AND MODIFIED ALGORITHMS (12-DB SNR INPUT) 47

TABLE 3.2 COMPLEXITIES AND AVERAGE COMPLEXITIES OF C MODULES OF VAD 50

TABLE 4.1 COMPLEXITIES AND AVERAGE COMPLEXITIES REDUCTION THROUGH PERFORMING PROGRAM-LEVEL OPTIMIZATION (12-DB SNR INPUT) 58

TABLE 4.2 INTRINSIC FUNCTIONS USED IN ALGORITHM 61

TABLE 4.3 COMPLEXITIES AND AVERAGE COMPLEXITIES REDUCTION THROUGH REORGANIZING MEMORY (12-DB SNR INPUT) 68

TABLE 4.4 COMPLEXITIES AND AVERAGE COMPLEXITIES REDUCTION THROUGH OPTIMIZATIONS (12-DB SNR INPUT) 69

TABLE 4.5 COMPLEXITIES AND AVERAGE COMPLEXITIES REDUCTION THROUGH OPTIMIZATIONS (6-DB SNR INPUT) 70

TABLE 4.6 COMPLEXITIES AND AVERAGE COMPLEXITIES REDUCTION THROUGH OPTIMIZATIONS (18-DB SNR INPUT) 71

TABLE 4.7 COMPLEXITIES AND AVERAGE COMPLEXITIES REDUCTION THROUGH OPTIMIZATIONS (INFINITE-DB SNR INPUT) 71

LIST OF ABBREVIATIONS

ALU	Arithmetic Logic Unit
ANSI	American National Standards Institute
API	Application Programming Interface
AR	Autoregressive
CCS	Code Composer Studio
CELP	Code-Excited Linear Prediction
CNG	Comfort Noise Generation
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
CSSU	Compare, Select, Store Unit
DAA	Data Access Arrangement
DC	Direct Current
DRR	Data Receive Register
DIP	Dual In-line Package
DSK	DSP Starter Kit
DSP	Digital Signal Processing or Digital Signal Processor
DSPLIB	DSP Library

DXR	Data Transmit Register
DTX	Discontinuous Transmission
FIR	Finite Impulse Responce
HPI	Host-Port Interface
HST	Host Channel
HWI	Hardware Interrupt
IDE	Integrated Development Environment
IDL	Idle Loop
ISR	Interrupt Service Routines
JTAG	Joint Test Action Group
LAR	Log Area Ratio
LED	Light Emitting Diode
LMS	Least-Mean-Square
LPC	Linear Predictive Coding
MAC	Multiply/Accumulate
McBSP	Multi-channel Buffered Serial Ports
MMSPE	Minimum Mean-Squared Prediction Error
PARCOR	Partial Correlation Coefficients
PCM	Pulse-coded Modulation

RTDX	Real-Time Data Exchange
SARAM	Single-Access RAM
SID	Silence Insertion Descriptor
SNR	Signal-to-Noise Ratio
SWI	Software Interrupt
TI	Texas Instruments
UART	Universal Asynchronous Receiver/Transmitter
VAD	Voice Activity Detection

CHAPTER 1

INTRODUCTION

1.1 Principle of VAD-CNG Algorithms

Many vocoder algorithms, especially low bit-rate speech coding algorithms, have been developed to meet the significantly increased demand for digital wireless as well as other packet voice communication systems [1]. Among these algorithms, ITU-T Recommendation G.729 (toll-quality under 8 Kb/s) [2] and its complexity-reduced version G.729A [3] are the most popular ones.

In voice communication, voice signals are generally classified into three non-overlapping types that are (1) active speech that carries very important information; (2) audible non-speech (inactive) that carries less useful but not necessarily useless information; and (3) silence (not audible) that carries no useful information [5]. Based on a well-known fact that in a full-duplex conversation, as high as 60% of the conversation in any one direction is inactive speech or background silence, voice activity detection (VAD) and comfort noise generation (CNG) algorithms [5]-[13], also called silence suppression algorithms, have been developed in order to avoid the transmission of packet signals during silent or inactive periods to reduce the transmission bandwidth. The use of these algorithms can lead to a higher number of users and an increase in the data throughput in packet voice communication systems.

A block diagram of a packet voice system using both VAD and CNG is shown in Figure 1.1 [4]. The input speech is passed to the VAD module that decides whether the speech is active or inactive. If the current frame is active, this frame of data is passed to a low bit-rate speech encoder and encoded and packed by it before being sent; if it is inactive, the transmission of this signal packet is suppressed and perhaps, a silence insertion descriptor (SID) packet, which contains several characteristic parameters of the background noise, is formed by a CNG encoder and sent to the far-end. A discontinuous transmission (DTX) algorithm is introduced to determine the frequency of the SID packet transmission during periods of inactive speech. At the receiving side, a CNG decoder uses the SID packets to generate comfort noises to fill in the gaps that the original silence or noise should occupy.

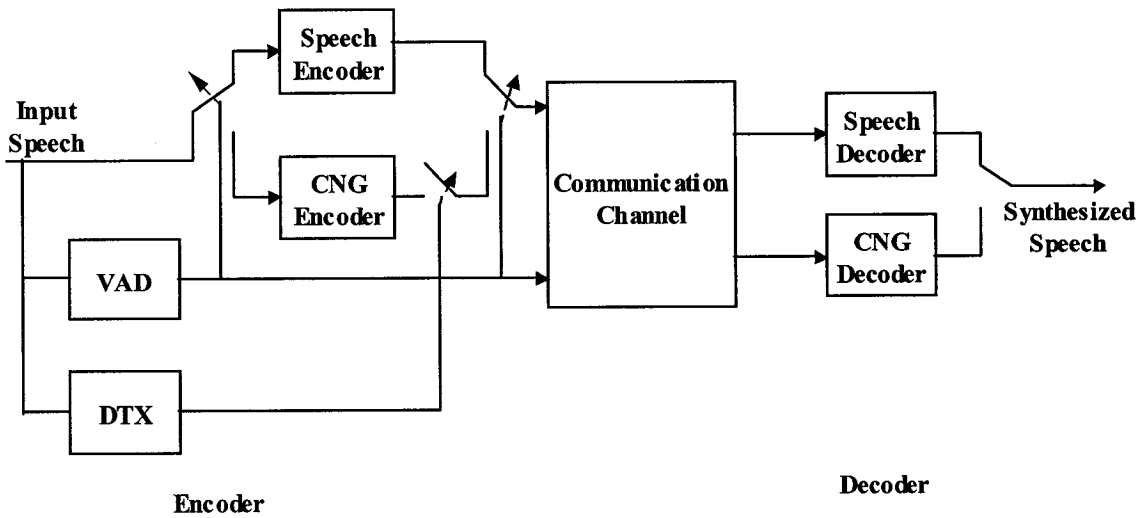


Figure 1.1. VAD/DTX/CNG block diagram [4]

Generally, the VAD at the encoder determines when speech is present and the CNG decoder fills in the gaps in the received packets by generating noise using the SID packets to make the synthesized speech neither irritating nor uncomfortable. Substantial reduction

in the transmission rate as well as in the transmission bandwidth is achieved through VAD-CNG algorithm during inactive periods, since the bandwidth used for transmitting the SID packets is negligible in comparison with the original bandwidth requirement.

1.2 Nortel VAD-CNG Algorithm

The Nortel fixed-point VAD-CNG algorithm [6] consists of three parts, namely, VAD, DTX and CNG, and employs linear predictive coding (LPC) and energy analysis to carry out the VAD. The output of the VAD module is either 1 or 0, indicating, respectively, the presence or absence of voice activity. If the VAD output is 1, a speech codec is invoked to code/decode the active voice frames. On the other hand, if the VAD output is 0, the DTX/CNG algorithms are used to code/decode the non-active voice frames.

1.2.1 VAD Algorithm

The Nortel VAD algorithm makes a voice activity decision on a frame-by-frame basis every 10 ms in accordance with the frame size of input speech, which is sampled at the rate of 8 KHz and segmented into frames of 80 samples. In this algorithm, a set of four parameters, namely, the peak energy, minimum energy, LPC gain, and the spectral non-stationarity, is extracted and based on these parameters, power likelihood, LPC gain likelihood, and non-stationarity likelihood, as well as a composite likelihood are calculated to make the voice activity decision. The block diagram for this VAD algorithm is shown in Figure 1.2 [18].

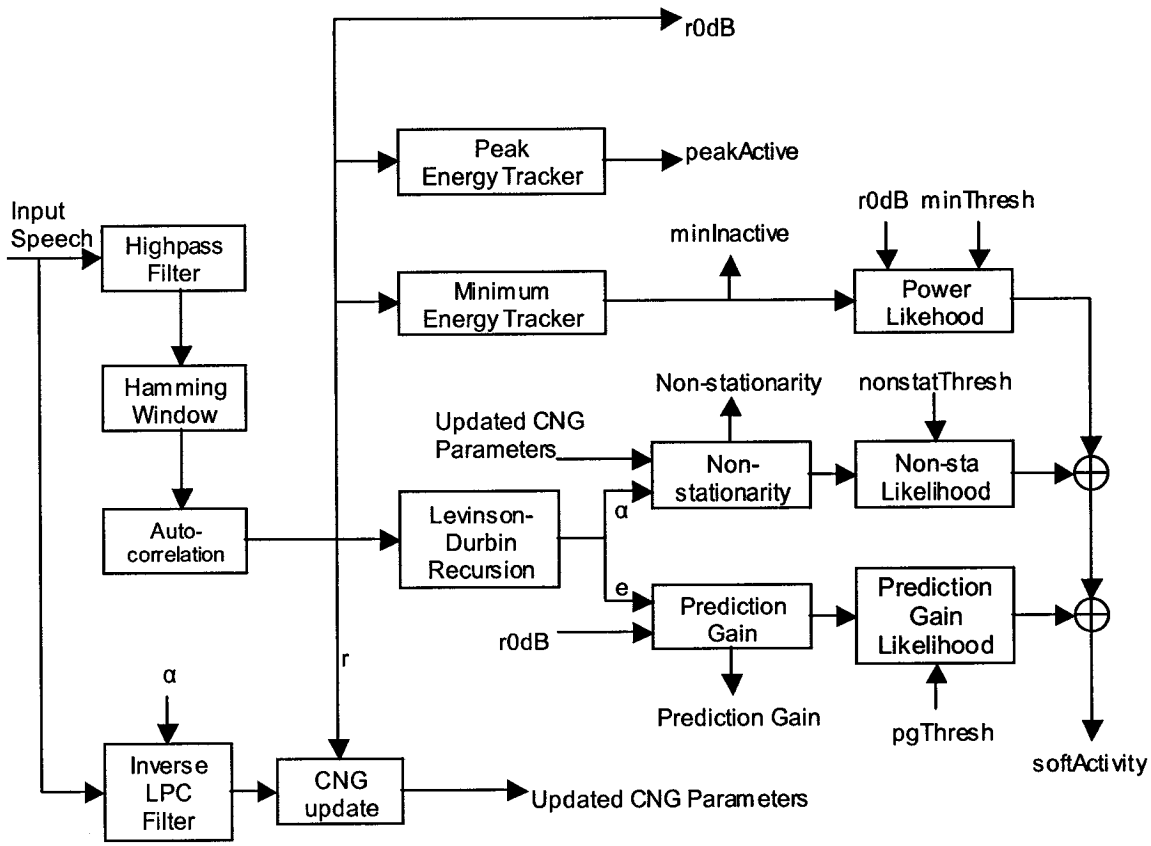


Figure 1.2. VAD block diagram [18]

1.2.1.1 LPC Analysis

The LPC analysis, which is widely used in voice signal processing algorithms [14]-[17], is employed is this VAD-CNG algorithm. The LPC analysis estimates, or gives the linear prediction of, a speech sample based on the assumption that the speech sample is modeled as the output of an all-pole glottal filter, given by [17]

$$H(z) = \frac{1}{1 + \sum_{i=1}^{p} a_i z^{-i}} \tag{1.1}$$

where $a_i (i = 1,2,\cdots, p)$ are the linear prediction coefficients, and p is the prediction order, which is taken as 10 in this algorithm. This all-pole model is also known as an *autoregressive* or AR model [15]. Hence, the speech signal $s(n)$ can be estimated as [17]

$$\hat{s}(n) = -\sum_{i=1}^{p} a_i s(n-i) \tag{1.2}$$

where $\hat{s}(n)$ is the predicted signal. The estimation residual signal, or the prediction error signal $e(n)$, is defined as

$$e(n) = s(n) - \hat{s}(n) = s(n) + \sum_{i=1}^{p} a_i s(n-i) \tag{1.3}$$

and can be regarded as the output of the following prediction error filter or inverse filter A(z):

$$A(z) = 1 + \sum_{i=1}^{p} a_i z^{-k} \tag{1.4}$$

with speech $s(n)$ as its input. The least-mean-square (LMS) method is used to choose a_i to minimize the mean energy in the error signal over a frame of speech data, or the mean-squared prediction error defined as [17]

$$E = \sum e^2(n) \tag{1.5}$$

It can be proved that the minimum mean-squared prediction error (MMSPE) could be obtained by solving the p-th order linear systems of equations [17]:

$$\boldsymbol{Ra} = -\boldsymbol{r} \tag{1.6}$$

where,

$$\boldsymbol{a} = (a_1, a_2, \cdots, a_p)^T \tag{1.6a}$$

$$\boldsymbol{r} = [r(1), r(2), \cdots, r(p)]^T \tag{1.6b}$$

and

$$\boldsymbol{R} = \begin{bmatrix} r(0) & r(1) & \cdots & r(p-1) \\ r(1) & r(0) & \cdots & r(p-2) \\ \vdots & \vdots & \ddots & \vdots \\ r(p-1) & r(p-2) & \cdots & r(0) \end{bmatrix} \tag{1.6c}$$

The autocorrelations $r(i)$ are calculated using the following equation:

$$r(i) = \sum_{n=0}^{N-i-1} s_w(n+l)s_w(n) \ , \qquad i = 0,1,2,\cdots,p \tag{1.7}$$

where $s_w(n), (n = 0,1,\cdots,N-1)$ are the windowed speech samples and N is the window size. The redundancy in this symmetric ($\boldsymbol{R}(i,k) = \boldsymbol{R}(k,i)$) and Toeplitz (all elements along a given diagonal being equal) matrix $\boldsymbol{R}$ allows one to use the more efficient Levinson-Durbin recursive procedure [15,16] instead of solving the matrix equation directly. One can refer to [15]-[17] for more details regarding the computation of $\boldsymbol{R}$, $\boldsymbol{r}$, $\boldsymbol{a}$, and the reflection coefficients $k_i (i = 1,2,\cdots,p)$, or the negatives of the reflection coefficients, the partial correlation (PARCOR) coefficients.

1.2.1.2 Prediction Gain and Non-stationarity

As shown in Figure 1.2, the short-term input speech is passed through a highpass filter to eliminate its direct current (DC) component and low-frequency hum, and then windowed by using a Hamming window of size 30 ms (or 240 samples). The LPC analysis is then applied to get the autocorrelations $\boldsymbol{r}$, the best linear prediction coefficients $\boldsymbol{a}$, and the prediction error e, where $\boldsymbol{a}$ and e are obtained from the Levinson-Durbin recursive procedure.

The prediction gain is defined as the prediction error normalized with respect to r (0), that is [4],

$$predictionGain = 10\log_{10}(\frac{r(0)}{e}) \tag{1.8}$$

A very large prediction gain implies that there are very strong spectral components or there is considerable spectral tilt. In either case, it is an indication that the signal is voice or that it is a signal that may be hard to regenerate with comfort noise [5].

Based on the common assumption that the background noise is relatively more stationary than the active speech signal from the point of view of the long-term average [5], the spectral non-stationarity is one of the best ways in identifying a speech from a noise and it is defined as [4]

$$nonstationarity = 10\log_{10}(\frac{r_v(0) + 2\boldsymbol{a}^T\boldsymbol{r} + \boldsymbol{a}^T\boldsymbol{R}_v\boldsymbol{a}}{r_v(0) + \boldsymbol{a}_v\boldsymbol{r}_v}) \tag{1.9}$$

where $r_v(i), (i = 0,1,2,\cdots, p)$ are obtained as the moving average values of $r(i)$ (see Section 1.2.2.1), $\boldsymbol{r}_v$ is their vector form as in (1.6b), and $\boldsymbol{R}_v$ is their matrix form as in (1.6c). The LPC coefficients $a_{vi}(i = 1,2,\cdots, p)$ are obtained from $r_v(i)$ through Levinson-Durbin recursion, whereas $\boldsymbol{a}_v$ is their vector form as in (1.6a). The denominator of (1.9) corresponds to the optimal prediction error obtained by filtering the windowed long-term averaged speech signal through the corresponding optimal LPC inverse filter, and the numerator corresponds to a prediction error obtained by filtering the same averaged speech signal through the LPC inverse filter which corresponds to the current frame's LPC analysis [4]. The spectral distance between these two prediction errors is measured in this way and a large value of this distance is an indication that the signal spectrum is changing rapidly [5]. It is also an indication of an active speech according to the assumption mentioned above.

1.2.1.3 Energy Analysis

Short-term energy is widely used in the VAD because of its efficiency and simplicity. The Nortel algorithm tracks both the peak energy and the minimum energy.

The peak energy is tracked by using a simple non-linear first-order filter, where the input is the speech energy (in dB) with a filter coefficient that is a function of the VAD state. In the original Nortel's floating-point algorithm, it is defined by the following difference equation [5]:

$$y(n) = \max(u(n), (1-\alpha)y(n-1) + \alpha u(n)) \tag{1.10}$$

where *u(n)* is the current value of the speech energy ("r0dB"), *y(n)* is the peak energy ("peakActive"), and α is chosen from a set of two possible constants. The larger value is used, if the current speech frame is declared active, otherwise the smaller one. The calculation is simplified in the fixed-point algorithm as [6]

$$y(n) = \begin{cases} u(n) & , if \quad u(n) \geq y(n-1) \\ y(n-1) - \alpha & , if \quad u(n) < y(n-1) \end{cases} \tag{1.11}$$

Similar to the peak tracker, the minimum energy ("minInactive") is tracked with the same function (1.10) except that the input is negative of the energy (in dB) [5,6].

If the speech energy is much less than the peak energy, the background noise is most likely inaudible or it is low enough to be declared as inactive; if the speech is much larger than the minimum background noise energy, this frame is declared as active [5].

1.2.1.4 Likelihood

In the Nortel algorithm, likelihood, a probability measure used by soft decisions, is applied when the hard decisions with fixed thresholds cannot make the decision with regard to a frame being active or inactive.

From the parameters obtained from the energy and LPC analyses, three likelihoods, namely the power likelihood (L_1), the prediction gain likelihood (L_2), and the non-stationarity likelihood (L_3), are computed as [5]:

$$L_j = \begin{cases} 0, & x \le th_0 \\ 1, & x \ge th_1 \\ \dfrac{x - th_0}{th_1 - th_0}, & otherwise \end{cases}, \qquad j = 1,2,3 \tag{1.12}$$

Each likelihood is calculated based on the value of the parameter x and a pair of thresholds th_0 and th_1 ("minThresh", "pgThresh", or "nonstatThresh"), where th_0 is the minimum threshold and th_1 the maximum threshold. In this way, a crude probability or likelihood of an active speech segment for a particular parameter is produced. By adding these three likelihoods together a composite likelihood L ("softActivity") is formed as

$$L = L_1 + L_2 + L_3 \tag{1.13}$$

Further details regarding the use of these likelihoods in the VAD decision rules may be obtained from [4-6].

1.2.2 CNG Algorithm

The CNG algorithm is designed to extract the noise parameters, to form the SID packets at the encoder, and to generate comfort noise by shaping the Gaussian noise spectrum with the decoded SID parameters at the decoder [18]. There are three factors that should be considered when the CNG is designed: (1) The background noise suppressed during the inactive periods should be reproduced at the receiver with a substantial fidelity to make the synthesized speech neither irritating nor uncomfortable. (2) The CNG operations should not affect the achievement of substantial bandwidth savings. (3) The SID parameter encoding should be efficient and at the same time, maintain the subjective quality of the generated comfort noise [5, 18].

1.2.2.1 CNG Encoder

The CNG encoder executes two tasks: (1) updates and extracts the set of characteristic noise parameters and (2) forms the SID packets whenever necessary.

During the inactive periods, the background noise power and the spectrum (the autocorrelation vector) are updated by averaging the short-term energy and spectrum with a non-linear filter given by [5]

$$y(n) = (1-\beta)y(n-1) + \beta u(n) \tag{1.14}$$

where *u(n)* and *y(n)* are, respectively, the current the updated parameters. The filter coefficient β is not a constant; it is a variable that is chosen from a set of two values depending on the difference between the current and smoothed parameters [5].

An SID packet includes the codewords of the quantized comfort noise power level and the log area ratios (LARs) of the reflection coefficients $k_i(i = 1,2,\cdots,p)$, which are the by-products of the Levinson-Durbin recursion. The comfort noise power level is simply quantized by its logarithm (dB). The noise spectrum information needs to be quantized before it can be packed into the SID packet and transmitted to the decoder. However, direct quantization of the prediction coefficients is not desirable [6]. The reflection coefficients are quantized instead. "While the reflection coefficients $k_i(i = 1,2,\cdots,p)$ are less sensitive to quantization than $a_{v_i}(i = 1,2,\cdots,p)$, $k_i(i = 1,2,\cdots,p)$ still cause difficulties when their magnitudes are near unity (i.e., reflection coefficients can be quantization-sensitive when they represent narrow-bandwidth poles)" [17]. With

appropriate nonlinear transformations expanding the region near $|k_i(i=1,2,\cdots,p)|=1$, the reflection coefficients are converted to LARs defined as [17]

$$LAR_i = \ln\frac{1+k_i}{1-k_i}, \qquad i=1,2,\cdots,p \tag{1.15}$$

for the quantization. The LAR is quantized using the mid-rise quantizer whose input-output characteristic is expressed by [17]

$$x_i = (i-\frac{L+2}{2})\Delta, \qquad i=2,3,\cdots,L \tag{1.16a}$$

$$y_i = (i-\frac{L+1}{2})\Delta, \qquad i=1,2,\cdots,L \tag{1.16b}$$

where L is the number of output levels and Δ is a fixed quantizer step size. Since p, the order of the prediction filter, is chosen as 10 in the algorithm, ten scalar quantizers with different values for L and Δ need to be designed for the ten LARs [6].

1.2.2.2 CNG Decoder

The CNG decoder performs the following four tasks: (1) unpacking of the received SID packets, (2) dequantization of the parameters (comfort noise power level and LARs) in the SID packets, (3) computation of the excitation gain based on the power level and CNG spectral parameters from decoded LARs, and (4) generation of the comfort noise by shaping the Gaussian noise spectrum with the decoded SID parameters.

As shown in Figure 1.3 [18], in order to produce the comfort noise, the Gaussian white noise is generated, scaled by the excitation gain calculated from the power level, and

passed through a LPC synthesis filter constructed using the LPC coefficients $acng_i (i = 1,2,\cdots, p)$ that are computed from the dequantized LARs.

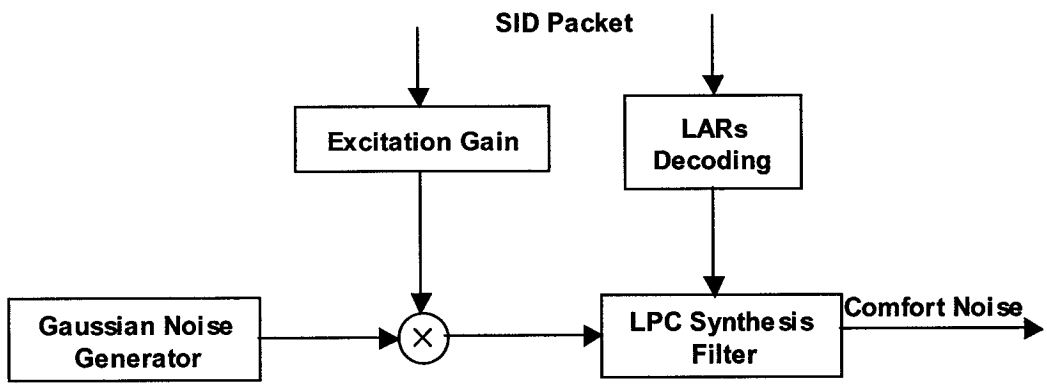


Figure 1.3. Comfort noise generation scheme [18]

The inversely quantized LARs are converted into CNG reflection coefficients $kcng_i (i = 1,2,\cdots, p)$ using [6]

$$kcng_i = \frac{e^{LAR_i} - 1}{e^{LAR_i} + 1}, \qquad i = 1,2,\cdots, p \tag{1.17}$$

The CNG synthesis filter coefficients $acng_i (i = 1,2,\cdots, p)$ are recovered from the reflection coefficients $kcng_i (i = 1,2,\cdots, p)$ by the recursion given by

$$a_i^{(i)} = kcng_i \tag{1.18a}$$

$$a_j^{(i)} = a_j^{(i-1)} + k_i a_{i-j}^{(i-1)}, \qquad 1 \le j \le i-1, \; i = 1,2,\cdots, p \tag{1.18b}$$

and the filter coefficients are given by

$$acng_j = a_j^p, \qquad j = 1,2,\cdots,p \tag{1.18c}$$

1.2.3 DTX Algorithm

The DTX algorithm determines whether or not an SID packet needs to be sent to the receiver for each inactive frame. The SID packet should be sent at rates as low as possible to save the transmission bandwidth, whereas the comfort noise generated at the receiver should be perceptually equivalent to the background noise at the transmitter [18].

In the Nortel algorithm, the SID packets are sent under the following conditions [4]:

(1) Every time a change from an active to an inactive speech is detected, an SID packet should be transmitted to the decoder to update the parameters that are employed to generate the comfort noise.

(2) During the inactive speech periods, an SID packet is sent every time when the number of counted silence frames reaches 50.

(3) During the inactive speech periods, an SID packet is sent when the difference of the energy level between the current and the previous residual signals exceeds 1 dB.

Through the DTX algorithm, the transmission of the SID packets will not affect the bandwidth efficiency or the quality of the restored background noise.

1.3 Modified Nortel VAD-CNG Algorithm

With an objective to achieve a reduction in the average computational complexity with very little or no loss in the performance, as well as to improve the performance and efficiency of the processing, some modifications have been introduced recently to the Nortel VAD-CNG algorithm through an investigation carried out in our laboratory [4]. The experimental results have shown that there is a reduction of more than 40% in the average computational complexity through these modifications, while the overall performance remains very close to that of the original Nortel algorithm. The modifications are discussed below briefly. However, interested readers are referred to Chapter 3 of [4] for more details.

For the sake of convenience, we will henceforth refer to the Nortel fixed-point VAD-CNG algorithm as the "original algorithm", whereas the modified algorithm in [4] as the "modified algorithm".

1.3.1 Revisions to Decision Rules

The original algorithm calculates all the LPC-related parameters, the prediction gain, the spectral non-stationarity, and the composite likelihood before making the VAD decisions, whether or not they are actually required, and thus, some unnecessary calculations are involved. Actually, a significant amount of the decisions are made by the energy-related parameters. Furthermore, even when the decisions are made by the LPC-related parameters, not all of these parameters are involved.

Hence, the calculations of these parameters, which are carried out during the processing of the signal in every frame in the original algorithm, are calculated only when they are needed in the modified algorithm. This modification reduces the amount of computations, and yet it has no negative influence on the performance of the algorithm [4].

1.3.2 Use of a Rectangular Window

A rectangular window is introduced in the modified algorithm to replace the Hamming window in the original algorithm. Generally speaking, a rectangular window is not suitable for speech signal processing [16]. However, if we are interested in voice activity detection alone, it is possible to use a rectangular window, since we do not need to restore the active speech signal. The spectral information obtained using a rectangular window may be less accurate than that using a Hamming window, but the VAD performance resulting by the use of a rectangular window can still be close to that of the original algorithm. By doing so, we can not only avoid the use of the window function, thus leading to a considerable reduction in the multiplication operations, but also make use of the overlap between the successive signal blocks to further reduce the computational load of the autocorrelation coefficients $r(i), (i = 1,2,\cdots, p)$ [4]. The test results have shown that an overall performance very close to that of the original algorithm can be achieved by the use of the modified algorithm employing a rectangular window after introducing some appropriate modifications [4].

1.3.3 Simplification of the Highpass Filter

It is well known that the frequency range from 200 to 5600 Hz contributes most to the speech perception, and this range matches the frequencies of the largest auditory sensitivity

and highest speech energy [17]. To remove the undesired low frequency components, such as the DC component and the low-frequency hum, a highpass filter given by [4]

$$H_{hp}(z) = \frac{(475/512) - (950/512)z^{-1} + (475/512)z^{-2}}{1 - (976/512)z^{-1} + (467/512)z^{-2}} \tag{1.19}$$

which is the same as the one used in G.729 except for the scaling factor [2], is employed in the original algorithm. However, in VAD applications, it is the DC component that we want to remove most. Therefore, we can use a relatively simple first-order filter to replace this second-order one to reduce the computational load. The filter used in the modified algorithm is given by [4]

$$H_{hp}(z) = \frac{1 - z^{-1}}{1 - (127/128)z^{-1}} \tag{1.20}$$

By comparing the frequency responses of the filters given by (1.19) and (1.20), we see that the second-order filter used in the existing Nortel algorithm has a larger cut-off frequency than that of the first-order filter, which means that the second-order filter can remove more low-frequency components. As for the DC component, however, the performance of the first-order filter is the same as that of the second-order filter [4].

1.3.4 Reduction in Frequency of SID Packets Transmission

As mentioned in Section 1.2.2.1, the CNG encoder updates the extracted noise parameters (the noise power and the autocorrelation vector) by a non-linear filter (see (1.14)) and forms the SID packets for every frame during the inactive periods. Also, as mentioned in Section 1.2.3, the SID packets are not sent in all frames during the inactive periods. It

means that unnecessary computations are involved in the preparation of an SID packet that is not transmitted during the current frame. In the modified algorithm, the extracted noise parameters are updated and an SID packet is formed only when it is required to be transmitted to the decoder. In this way, the unnecessary calculations are avoided. Considering that the background noise is relatively more stationary than an active speech signal from the point of view of the long-term average, and in view of using the running average scheme, the frequency of this updating can be appropriately reduced with little effect on the performance of the algorithm.

1.3.5 Other Modifications

Besides the modifications mentioned above, some other modifications, such as the adjustments of some constants, coefficients, and thresholds, are carried out to improve the performance of the modified algorithm and correct the improper processing in the original one [4].

1.4 Motivation and Scope of the Thesis

An algorithm needs a real-time implementation, rather than just simulations, to show its practical value. Not only is its performance important but also its implementational complexity when it is implemented, considering the real-time requirement and the product cost.

In this research, the modified algorithm is implemented on a TMS320C5402DSK DSP board to build a real-time system. Considering that the VAD-CNG algorithm is designed to be used as a part of a packet voice communication system, which consists of many integral

parts, such as the coder, decoder, echo canceller, and voice enhancement, and all these parts need to work together in real-time, there exists a need to reduce the complexity of this algorithm as much as possible. To achieve this objective, some optimizations are carried out to further reduce the implementational complexity of the modified algorithm. The need for the complexity reduction arises not only from the requirement of a real-time implementation, but also from the fact that the reduction in implementational complexity always leads to a reduction in the CPU frequency requirement, power, and memory space, i.e., a lower complexity means a lower product cost. Several DSP/BIOS real-time analysis tools are used to test the effects of the implementation and the optimizations.

The optimizations for reducing the complexity are carried out at the implementation level rather than at the algorithm level. Specifically, the implementational optimizations, such as replacing some computationally intensive C routines with assembly routines, making use of the assembly-optimized functions from the DSP Library and intrinsic functions, and reorganizing the memory, rather than modifying the algorithm itself, are undertaken.

1.5 Organization of the Thesis

In Chapter 2, an overview of the TMS320C5402DSK DSP board is given. The direct implementation of the modified algorithm is presented in Chapter 3. In Chapter 4, certain optimizations to the implementation are proposed and some tests are carried out to evaluate the effects of the optimizations. Based on the optimized implementation, a real-time demonstration system using real I/O peripheral devices is built in Chapter 5. Finally, conclusions as well as suggestions for future work are presented in Chapter 6.

CHAPTER 2

AN Overview OF TMS320C5402 DSP KIT

Due to advances in VLSI technology, programmable DSP devices are becoming increasingly available, affordable, and, therefore, popular in the industry for the design of DSP products [35]. TMS320C54xx from Texas Instruments (TI) is a widely applied family of DSP devices.

The modified algorithm is implemented on a TMS320C5402DSK (DSP starter kit) board, which is a DSP-developing tool from Texas Instruments (TI), providing a low-cost, standalone 54x development platform that enables DSP designers and programmers to evaluate and develop applications for the C54x DSP. The DSK also serves as a hardware reference design [19]. We will now describe its hardware, software and real-time analysis tools in this chapter.

2.1 Hardware Overview

The hardware of TMS320C5402DSK includes:

- TMS320VC5402 DSP
- CY37128 CPLD (complex programmable logic device)
- External memory
- Microphone/speaker interface
- IEEE-1284 DB-25 compatible parallel-port type host interface
- Power supply

- RS-232 UART (universal asynchronous receiver/transmitter) data interface
- Telephone DAA (data access arrangement) interface
- JTAG (joint test action group) emulation and host port interface

The hardware tools of this kit specifically used in the implementation are discussed in the following sections.

2.1.1 TMS320VC5402 DSP

The TMS320VC5402DSP is a fixed-point digital signal processor (DSP) in the TMS320 DSP family, which is designed to meet the specific needs of real-time embedded applications. It combines an advanced modified Harvard architecture (with one program memory bus, three data memory buses, and four address buses), a 100 MHz central processing unit (CPU) with application-specific hardware logic, on-chip memory (4K words of ROM and 16K words of DARAM (dual-access RAM)), on-chip peripherals, and a highly specialized instruction set [20, 21, 22]. The CPU consists of (1) a 40-bit arithmetic logic unit (ALU), including a 40-bit barrel shifter and two independent 40-bit accumulators, (2) a 17-bit × 17-bit parallel multiplier coupled to a 40-bit dedicated adder for nonpipelined single-cycle multiply/accumulate (MAC) operations, (3) a compare, select, store unit (CSSU) for the add/compare selection of the Viterbi operator, (4) an exponent encoder to compute the exponent of a 40-bit accumulator value in a single cycle, and (5) two address generators, including eight auxiliary registers and two auxiliary register arithmetic units.

2.1.2 CY37128 CPLD

The DSK uses a Cypress CY37128 CPLD to implement the required logic of the board and to provide control and status interfaces for the DSP software [19]. The CPLD provides the following functions:

- Reset control
- DSP memory-mapped control/status registers
- Peripheral decoding (UART and control registers)
- LED (light emitting diode) control and DIP (dual in-line package) switch status
- Host and DSP interrupt control
- Data transceivers control
- User options

The CY37128 CPLD is a 3.3 V (5 V tolerant), 160-pin device that provides 128 macrocells, 128 I/O pins, a 10 ns pin-to-pin delay, and 125-MHz maximum clock frequency. The device is EEPROM-based and is in-system programmable via a dedicated JTAG interface (a 10-pin header on the DSK) [19].

2.1.3 External Memory

Besides the on-chip memory, the DSK provides 64K × 16-bits (word) of SARAM (single-access RAM) and 256K × 16-bits of FLASH memory. The difference between SARAM and DARAM is that each DARAM block can be accessed twice per machine cycle and the CPU and peripherals, such as a multi-channel buffered serial port (McBSP) and host-port interface (HPI), can read from and write to a DARAM memory address in the same cycle,

whereas each SARAM block can be accessed only once per machine cycle [20]. The DSK also provides two expansion connectors to expand the memory and can address up to 16 × 64K words [19].

2.1.4 Microphone/Speaker Interface

Two 3.5 mm audio jacks are used as microphone and speaker interfaces and connected to the McBSP1 of the TMS320VC5402DSP through a TLC320AD50, which is used as a codec to digitize the input analog speech signal. The McBSP1 is a high-speed, full-duplex multi-channel buffered serial port, which allows continuous data streams between the TMS320VC5402DSP and the TLC320AD50 [24].

2.2 Software Overview

The DSK application-related software can be divided into (1) a host software, which supports C54x DSK board control, DSP application loading and execution, device configuration, status display, communication between the host and DSK, and board confidence tests, and (2) a DSP target software, which provides application programming interface (API) functions to develop applications that can control and operate the on-board peripherals [19]. Both the host software and the DSP target software are integrated into the Code Composer Studio (CCS), a graphic interface development tool providing a fully integrated development environment (IDE) supporting TI TMS320 DSP platforms [25].

2.2.1 Code Composer Studio

The CCS integrates all the host and target tools in a unified environment, including the DSP/BIOS kernel, editor, code-generation tools, debugger, and Real-Time Data Exchange

(RTDX) technology, in order to simplify the DSP system configuration and application design. As shown in Figure 2.1 [26], the CCS consists of the following items:

- Code-generation tools, including a C/C++ complier, an assembler, and a linker
- CCS integrated development tools, including an editor, a configuration tool, a project management tool, and a debugger
- DSP/BIOS kernel and its API [27], which control and operate the on-board peripherals
- RTDX API, which handles the real-time data exchange between the host and DSK

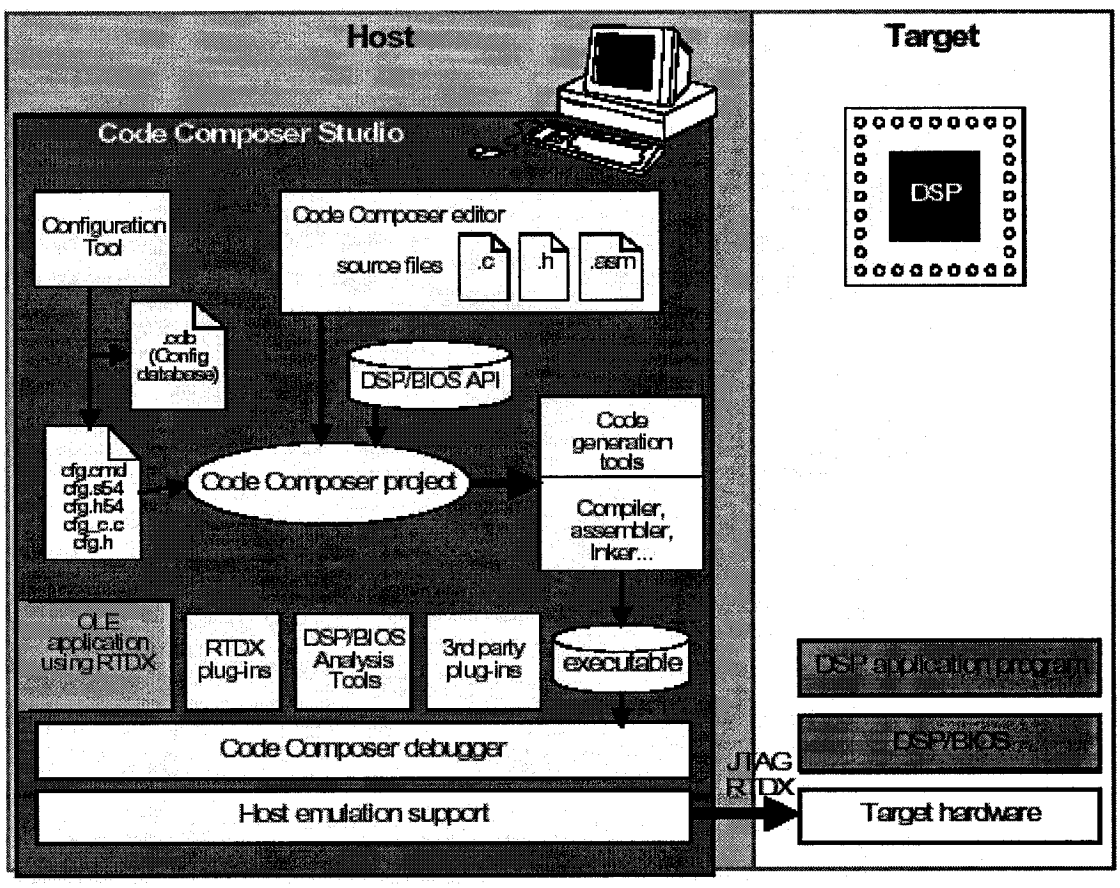


Figure 2.1. Block diagram of CCS [26]

CCS supports iterative program development cycles. We can create the basic framework for an application first, test it using analysis tools and then modify the framework, if necessary. A sample DSP development cycle includes the following steps, with the possibility of a step or a group of steps being iterated [26]:

1. Use the configuration tool to create objects for the program to use.
2. Save the configuration file to generate files to be included in the program.
3. Write a framework for the program using the editor. C, C++, assembly, or a combination of the languages can be used.
4. Create a project for the program using the project management tool.
5. Compile and link the program using the complier, assembler, and linker to generate executable code.
6. Test program behaviors using the debugger, DSP/BIOS analysis tools, and RTDX.
7. Repeat steps 1-6 until the program runs correctly. It is possible to add functionality and make changes to the basic program structure.
8. When production hardware is ready, modify the configuration file to support the production board and test the program on the board.

2.2.2 DSP/BIOS

DSP/BIOS is a scalable real-time kernel of CCS. It is designed for applications that require real-time scheduling and synchronization, host-to-target communication, or real-time instrumentation. DSP/BIOS provides preemptive multi-threading, hardware abstraction, real-time analysis (which will be discussed in Section 2.3) and configuration tools [26].

2.2.2.1 Configuration Tool

The Configuration Tool is a visual editor with an interface similar to the Windows Explorer [26], as shown in Figure 2.2. It allows one to create and configure the DSP/BIOS objects statically used by the program. These objects include software interrupts (SWI), hardware interrupts (HWI), I/O streams, and event logs. This tool can also be used to configure memory, thread priorities, and interrupt handlers. The objects created and properties set are used by the DSP/BIOS API at run-time [19]. When this configuration file is saved, the configuration tool creates an assembly source and header files and a linker command file to match the settings. When the application is built, these files are linked with the application programs [26].

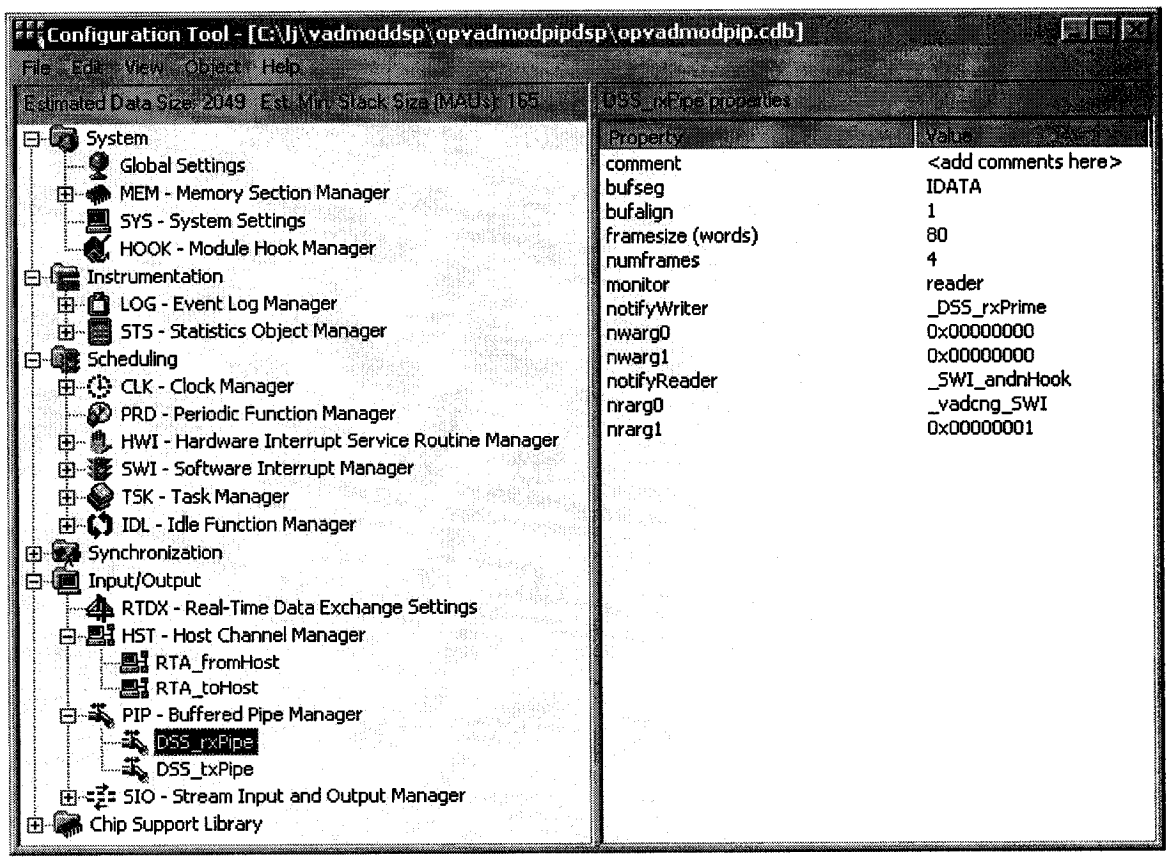


Figure 2.2. Configuration tool

2.2.2.2 Thread Scheduling

Many real-time DSP applications must perform a number of functions at the same time, often in response to external events such as the availability of data or the presence of a control signal. The functions performed, as well as when they are performed, are both important. These functions are called threads in DSP/BIOS, which include any independent stream of instructions executed by the DSP, such as interrupt service routines (ISR) or function calls [26].

DSP/BIOS enables applications to be structured as a collection of threads, each of which carries out a modularized function. Multithreaded programs run on a single processor by allowing higher-priority threads to preempt the lower-priority threads and by allowing various types of interaction between the threads, including blocking, communication, and synchronization [26].

DSP/BIOS provides support for several types of program threads with different priorities. Each thread type has different execution and preemption characteristics. The thread types (from highest to lowest priority) are as follows:

- Hardware interrupts (HWI), which includes clock (CLK) functions
- Software interrupts (SWI), which includes periodic (PRD) functions
- Tasks (TSK)
- Background thread (IDL)

HWIs have the highest priority among the four thread types and are triggered either by on-chip peripherals or by devices external to the DSP in response to external asynchronous

events that occur in the DSP environment. They should be used for application tasks that need to run at very high frequencies. An HWI function (also called an ISR) is executed after a hardware interrupt is triggered in order to perform a critical task that is subject to a hard time deadline. Using the Hardware Interrupt Service Routine Manager in the Configuration Tool, we can configure the ISR for each hardware interrupt in the DSP. An ISR can be written using assembly language, C, or a combination of both; however, it is usually written in assembly language for efficiency [26, 28].

The SWIs are patterned after the HWIs, with the HWIs triggered to call ISRs, whereas the SWIs triggered to call the SWI functions from the program. Software interrupts provide additional priority levels between the HWIs and TSKs. The SWIs handle threads subject to time constraints that preclude them from being run as TSKs, but whose deadlines are not as severe as those of HWIs are. SWI objects can be created and configured either dynamically by using DSP/BIOS API or statically by using the Software Interrupt Manager in the Configuration Tool [26].

TSKs have higher priority than the background thread and lower priority than the software interrupts. Tasks differ from software interrupts in that they can be suspended during the execution until necessary resources are available. TSK objects can also be created and configured either dynamically or statically [26].

Background thread executes the idle loop (IDL) at the lowest priority in a DSP/BIOS application. After main returns, a DSP/BIOS application calls the start-up routine for each DSP/BIOS module and then falls into the idle loop. The idle loop is a continuous loop that calls all functions for the IDL objects, which is created and configured by using the Idle

Function Manager in the Configuration Tool. The idle loop runs continuously except when it is preempted by higher-priority threads. Only functions that do not have hard time deadlines, such as communication between the target and the DSP/BIOS analysis tools, should be executed in the idle loop [26].

2.2.2.3 Memory Management

The Memory Section Manager of the Configuration Tool manages named memory segments that correspond to physical ranges of memory. It allows one to specify the memory segments required to locate the various code and data sections of a DSP/BIOS application [19].

TABLE 2.1. NAMES OF MEMORY SEGMENTS

Segment	Description
IDATA	Internal (on-chip) data memory
EDATA	External (off-chip) data memory
IPROG	Internal (on-chip) program memory
EPROG	External (off-chip) program memory
USERREGS	Page 0 user memory (28 words)
BIOSREGS	Page 0 reserved registers (4 words)
VECT	Interrupt vector segment

The standard memory segment names used by DSP/BIOS are listed in Table 2.1. It is also possible to create one's own memory segments with different names and properties, delete memory segments, rename memory segments, or change the properties of the

existing memory segments, such as the starting address, the length, and whether a memory segments is used as data space or program space.

2.2.2.4 Input and Output Management

Input and output for DSP/BIOS applications are handled by the stream, pipe and host channel objects, which are created and configured in the Configuration Tool. We now describe the pipe (PIP) and host channel (HST) modules, which are used in the implementation.

Pipes are designed to manage block I/O (also called stream-based or asynchronous I/O). Each PIP object, which is created and configured by Buffered Pipe Manager in the Configuration Tool, maintains a buffer divided into a fixed number of fixed length frames, specified by the "numframes" and "framesize" properties. All I/O operations on a pipe deal with one frame at a time [26].

As shown in Figure 2.3 [26], a pipe has two ends. The writer end is where the program writes the frames of data. The reader end is where the program reads the frames of data.

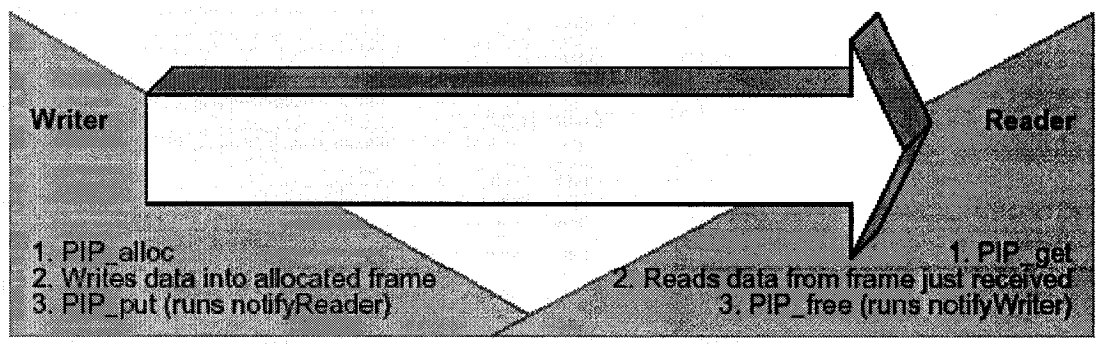


Figure 2.3. PIP module [26]

Data notification functions ("notifyReader" and "notifyWriter") are performed to synchronize the data transfer. These functions are triggered when a frame of data is written or read to notify the program that data is available or a frame is free. When a frame of data is written into an allocated frame, the writer end calls PIP_put assembly macro to put the frame into the pipe. The function notifyReader is then triggered to notify the program that the data is available and the reader end should call PIP_get assembly macro to read the data; Similarly, when a frame of data is read out from the pipe, the reader end calls PIP_free assembly macro to clear and recycle back this frame to the pipe. The function notifyWriter is then triggered to notify the program that a frame is free and the writer should call PIP_alloc assembly macro to allocate the next empty frame.

The Host Channel Manager manages the HST objects, which allow an application to stream data between the target and the host. The Host channels are configured for input or output, where input streams read data from the host to the target and output streams transfer data from the target to the host [26].

The input and output channels are, respectively, bound to the corresponding input and output files on the PC host by using the Host Channel Control dynamically, as shown in Figure 2.4.

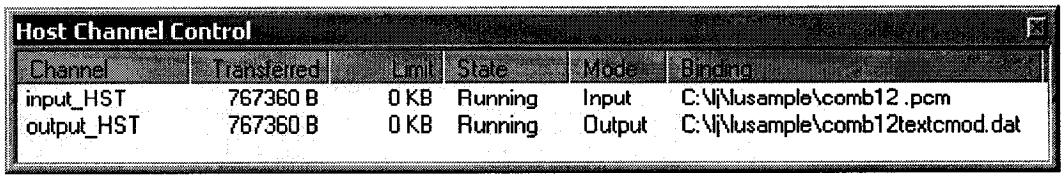


Figure 2.4. Host channel control

Each host channel is internally implemented using a PIP object. To use a particular host channel, the program uses HST_getpipe assembly macro to get the corresponding PIP object and then transfers data by calling the PIP_get and PIP_free operations (for input) or PIP_alloc and PIP_put operations (for output) [26].

2.3 Real-time Analysis Tools

Real-time analysis is the analysis of data acquired during real-time operation of a system. The intent is to determine easily whether or not the system is operating within its design constraints, is meeting its performance targets, and has scope for further development. DSP/BIOS provides explicit and implicit ways to perform real-time program analysis. Communication between the target and the DSP/BIOS analysis tools is performed within the background idle loop. This ensures that the DSP/BIOS analysis tools do not interfere with the program's processing. If the target CPU is too busy to perform the background processes, the DSP/BIOS Analysis Tools stop receiving information from the target until the CPU is available. These mechanisms are designed to have minimal impact on the application's real-time performance [26]. The analysis tools, Execution Graph, Statistics View, and CPU Load Graph used in the implementation are described in the following sections.

2.3.1 Execution Graph

The Execution Graph is a special graph used to display the information about SWI, PRD, TSK, SEM (Semaphore) and CLK processing. The Execution Graph window, as shown in Figure 2.5, shows the execution information as a graph of the activity of each object. CLK and PRD events are shown to provide a measure of time intervals within the Execution

Graph. Assertions are indications that either a real-time deadline has been missed or an invalid state has been detected.

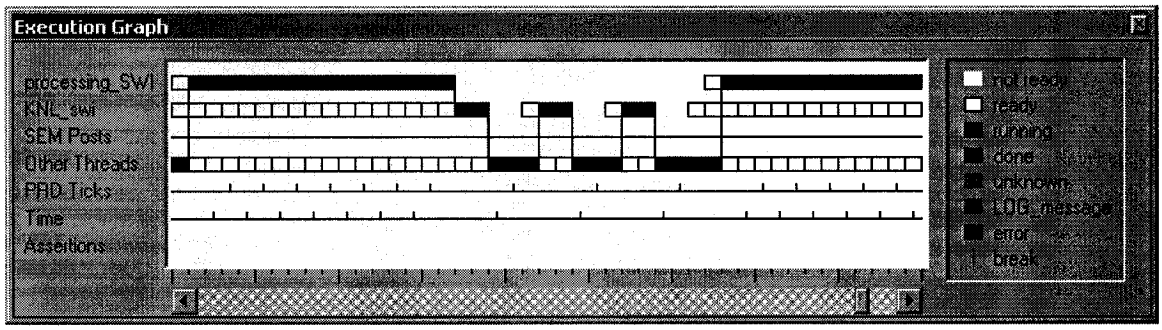


Figure 2.5. Execution Graph

2.3.2 CPU Load Graph

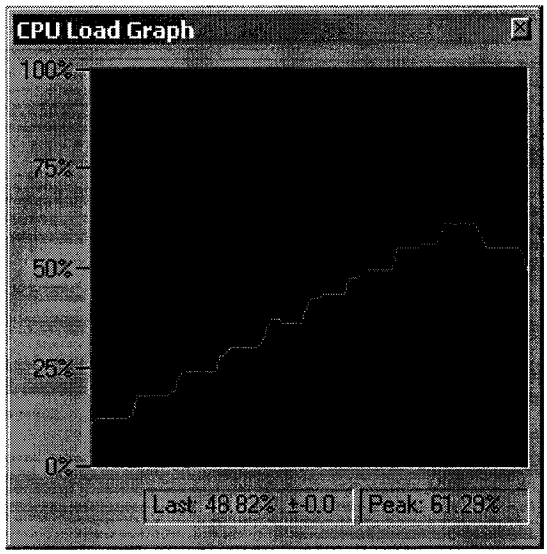


Figure 2.6. CPU Load Graph

The CPU load is defined as the percentage of instruction cycles that the CPU spends performing application works. The CPU Load Graph, as shown in Figure 2.6, illustrates the CPU load in real-time. All CPU activity is divided into a work time and an idle time. To measure the CPU load over a time interval T, we need to know how much time during that

interval is spent doing the application works (t_w) and how much of it is the idle time (t_i). From this, the CPU load is calculated as follows [26]:

$$CPUload = \frac{t_w}{T} \times 100\% = \frac{t_w}{t_w + t_i} \times 100\% \quad (2.1)$$

2.3.3 Statistics View

Figure 2.7 shows the values of STS (statistics) objects, which are created and configured by using the Configuration Tool. The first line shows the number of times an SWI is triggered and the number of instructions performed during this SWI's execution. DSP/BIOS supports such statistics automatically. This is called implicit instrumentation. We can also use the explicit instrumentation to gather other real-time statistics to study the performance of different parts of an application by bracketing appropriate sections of the program with the STS_set and STS_delta operations as

```
STS_set(&stsObj, CLK_gethtime());
The segment of the code under test
STS_delta(&stsObj, CLK_gethtime());
```

Statistics View

STS	Count	Total	Max	Average
processing_SWI	9069	5065101810 inst	914620 inst	558507.20
processingLoad_STS	9069	2519458733 ints	455850 ints	277809.98

Figure 2.7. Statistics View

The STS_set saves the value of the CLK_gethtime as the content of the previous value field (the set value) in the STS object. The STS_delta subtracts this set value from the new value passed by the second CLK_gethtime. The result is the difference between the time

recorded before the code segment started and after it is completed; i.e., the time taken to execute it. For a CPU with certain frequency (100 MHz), this time corresponds to a certain number of instructions (multiplying the time by 100 M).

In Figure 2.7, **Count** is the number of times an SWI is triggered or a code segment executed, **Total** is the arithmetic sum of the instructions executed for an SWI or a code segment, **Max** is the maximum number of instructions used to execute an SWI or a code segment, and **Average** is the average number of instructions executed for an SWI or a code segment.

The Statistics View is the main tool to check if an implementation meets the real-time requirement and reveal the implementational complexity.

2.4 Conclusion

TMS320C5402DSK provides a development platform for DSP designers and programmers. Its hardware and software structures are designed to support the development of real-time embedded applications. The integrated development environment (IDE) makes the development much easier and faster. The real-time analysis tools can provide the analysis of data acquired during real-time operation of an application.

CHAPTER 3

DIRECT IMPLEMENTATION OF THE MODIFIED ALGORITHM

In this chapter, the modified algorithm is implemented directly on a TMS320C5402DSK DSP board using an on-board test vector, and then using the host channels (HST module) and the pipes (PIP module), provided by DSP/BIOS. To evaluate the performance of the implementations, the real-time analysis tool, Statistics View, and a speech sample database provided by the standard TIA/EIA/IS-727 are employed.

3.1 C Modules of VAD Algorithm

To schedule the program threads in the implementation (see Section 2.2.2.2) and prepare for the optimizations in the next chapter, it is necessary to analyze the main C modules of the modified algorithm and find which modules consume the most computational resources.

As described in Section 1.2, the VAD-CNG algorithm consists of three parts, namely, VAD, CNG and DTX, which correspond to the *vad* and *cng* modules in the program, whereas the DTX part is integrated into the *vad* module. The *vad* module, which makes a voice activity decision on a frame-by-frame basis every 10 ms (millisecond), consumes about 69% of the computational resources of the entire algorithm, while the *cng* module consumes 31% (see Section 3.5). Our research will focus on the *vad* module.

The *ppVadInit*, *ppCngInit* and *ppVadCfg* C modules are executed only once in the program to perform the initialization and configuration of the program before processing the input data and contribute nothing to the complexity of the algorithm because there is no real-time requirement for them.

It will be shown in Section 3.5 that the *VadAnalysis* module (ppVAnal.c), which performs almost all of the tasks of the *vad* module except the reading of the input data, accounts for more than 99% of the complexity of the *vad* module. The C modules of *vad*, actually of *VadAnalysis*, is shown in Figure 3.1 and described below briefly. Please refer to [6] for more details. Notice that the *vad* module includes not only the VAD algorithm, but also one part of the CNG algorithm, the CNG encoder, which updates the noise parameters and forms the SID packets at the encoder.

The input speech is passed through a highpass filter by calling the *hpfilter* module (iir_df2.c). Then the LPC analysis is applied to the highpassed signal. The LPC analysis involves two C modules, *autocorr* (autocorr.c) and *levdurb* (levdurd.c), which execute the autocorrelation calculation and Levinson-Durbin recursion functionalities of the VAD algorithm.

The VAD decision is implemented with the following modules:

- *lintodb* (lintodb.c), which converts a linear value to its dB format, calculates the logarithm of $r(0)$ ("r0dB") and the prediction gain ("predictionGain").
- *d_lr* (d_lr.c), calculates the non-stationaity ("nonstationaity").
- *softDecision* (ppVAnal.c), calculates the likelihoods and makes the VAD decision.

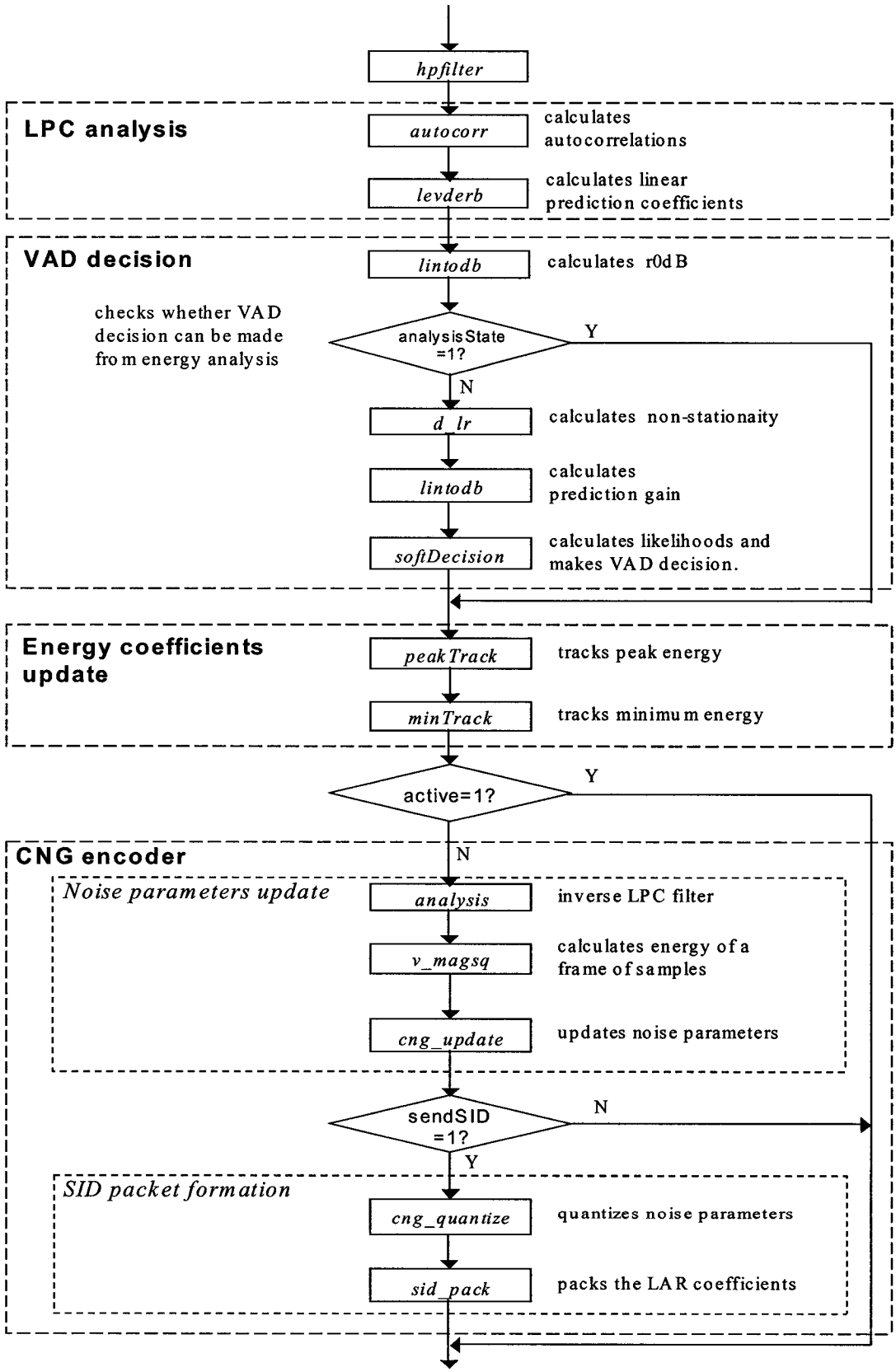


Figure 3.1. Flowchart of the *vad* module

The energy coefficients are updated with two modules, *peakTrack* (peaktrack.c), which tracks the peak energy ("peakActive"), and *minTrack* (mintrack.c), which tracks the minimum energy ("minInactive").

If a frame of data is determined as inactive (active = 0), the CNG encoder is applied to update the noise parameters and form the SID packets. The noise parameters should be updated with the following modules:

- *analysis* (analysis.c), which is an all-zero inverse LPC filter, calculates the residual signal $e(n)$.
- *v_magsq* (v_magsq.c), calculates the energy of a frame of samples.
- *cng_update* (cng.c), updates the noise parameters.

If a SID packet should be sent (sendSID = 1), the SID packet should be prepared using the following modules:

- *cng_quantize* (cng_quan.c), quantizes the noise parameters.
- *sid_pack* (sid_pack.c), packs the LAR coefficients to form the SID packet.

After the implementation, we will try to find the most computation-consuming modules in *vad* and apply certain optimizations on them in the next chapter.

3.2 Test Vectors and Complexity

To test the performance of the implementation, some voice vectors with different signal-to-noise ratios (SNRs) generated from the database of the standard TIA/EIA/IS-727 are used as the input to the algorithm during the test. Two metrics, complexity and average complexity, are introduced to evaluate the performance of the implementation.

3.2.1 Test Vectors

Standard TIA/EIA IS-727, *TDMA Cellular/PCS - Radio Interface - Minimum Performance Standards for Discontinuous Transmission Operation of Mobile Stations*, is designed to give the performance requirements for VAD to be used with enhanced full-rate vocoder with CNG, which is applied to mobile stations operating in the DTX mode [29]. This standard provides 10 speech data files and 4 background noise files which can be used to generate binary 16-bit pulse-coded modulation (PCM) test data files with different SNRs through the software tools provided by the standard. For our tests, 6-dB, 12-dB, 18-dB and infinite-dB SNR voice files are generated as test vectors.

If the test vector is loaded to the **EDATA** segment of the DSP board, a very short test vector is employed because of the space limitation of the on-board memory. The **EDATA** is a 32K-word segment on external SARAM; therefore, a 4-second 12-dB test vector (32000 words) is generated and stored in an assembly file, samples.asm, which will be loaded on the **EDATA** segment.

3.2.2 Complexity and Average Complexity

To evaluate the performance of the implementation, two metrics, complexity and average complexity, are introduced.

Complexity is defined as the maximum number of million instructions that might be executed per second for a program or function. The unit of complexity is million instructions per second (MIPS). In the Nortel algorithm, the sampling rate is chosen as 8000 HZ and the frame length as 80, which means the algorithm processes the data of one

frame every 10 ms for full-duplex transmission. So the complexity of this algorithm should be calculated as

$$Complexity = \frac{Max}{10ms * 10^6} = Max * 10^{-4} \ MIPS \tag{3.1}$$

where *Max* is the maximum number of the instructions executed in a cycle, which could be found in the **Max** column of the Statistic View window of the CCS.

Similarly, the average complexity defined as the average number of million instructions executed per second for a program or function and the unit is also MIPS. It is calculated as

$$Average\ Complexity = \frac{Average}{10ms * 10^6} = Average * 10^{-4} \ MIPS \tag{3.2}$$

where *Average* is the average number of the instructions executed in a cycle, which could also be found in the **Average** column of the Statistic View window of the CCS.

3.3 Implementation Using an On-board Test Vector

The modified algorithm is first implemented using a 4-second on-board test vector as the input data. This test vector is stored in an assembly file, samples.asm, which will be loaded to the **EDATA** segment on the off-chip memory of the DSP board.

3.3.1 Implementation Process

In a full-duplex transmission system, both the *vad* and *cng* modules are executed once every 10 ms, since the sampling rate is chosen as 8000 Hz and the frame length as 80. Using the scheduling module of the Configuration Tool, two periodical SWIs, **vad_SWI**

and **cng_SWI**, with the same priority are created to trigger the *vad* and *cng* modules, respectively. Both the periods of these two SWIs are set to 10. By default, the PRD Manager uses the CLK Manager (both in the Configuration Tool) to drive the execution of periodical SWI. The CLK Manager makes a clock interrupt, which is a HWI, trigger a PRD tick each ms. The SWIs will be triggered each time the PRD ticks reaches 10. In this way, these two periodical SWIs run their functions, *vad* function and *cng* function, every 10 ms.

The total size of all program sections is 12.446K words, whereas that of all data sections is 36.591K words. Therefore, it is impossible to load the entire program to the on-chip memory (16K-word DARAM and 4K-word ROM). The **.text** section, which includes the executable code, is placed in the **EPROG** segment on the off-chip memory. The **.data** section, which includes the 32000-word test vector, is loaded to the **EDATA** segment on the off-chip memory. The other sections are placed in the **IPROG** or **IDATA** segments on the 16K DARAM according to whether they are program or data sections. **IPROG** and **IDATA** are two 8K-word memory segments defined on the on-chip DARAM while **EPROG** and **EDATA** are two 32K-word segments on external SARAM. 4K-word memory space of **EPROG** is mapped onto the on-chip ROM. Please refer to Figure 4.2 (a) for the memory map of the implementation.

The *ppVadInit*, *ppCngInit* and *ppVadCfg* modules are included in the main function, which performs the initializations and configurations, and then falls into the infinite idle loop till **vad_SWI** or **cng_SWI** is triggered.

Five STS objects are created using the Statistics Object Manager in the Configuration Tool to use the explicit instrumentation (see Section 2.3.3) to gather real-time statistics to

study the performance of the modules we want to analyze. The Statistics View of this implementation is shown in Figure 3.2.

Statistics View

STS	Count	Total	Max	Average
vad_SWI	400	57874690 inst	266426 inst	144686.73
cng_SWI	400	15070980 inst	259386 inst	37677.45
VadAnalyze_STS	400	56872092 inst	263918 inst	142180.23
hpfilter_STS	400	11825810 inst	29566 inst	29564.53
autocorr_STS	399	32240214 inst	92412 inst	80802.54
levdurb_STS	267	4698242 inst	18736 inst	17596.41
analysis_STS	72	2287440 inst	31770 inst	31770.00

Figure 3.2. Statistic View of the implementation using an on-board test vector

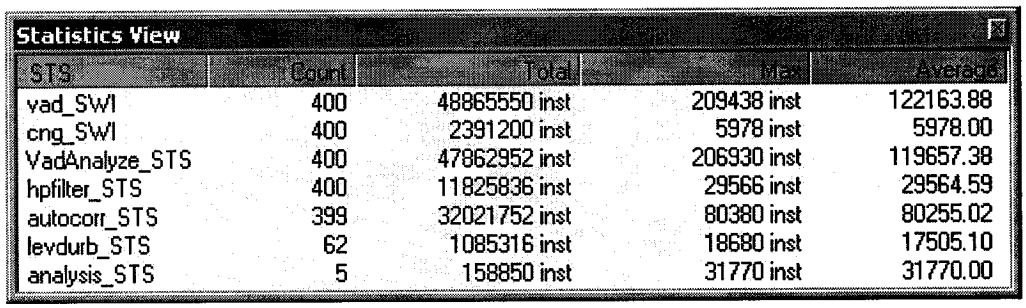
From Figure 3.2, we can show by using (3.1) that the complexity of the *vad* module is 26.64 MIPS and that of the *cng* module 25.94 MIPS. Also, the complexity of the entire modified algorithm can be shown to be 52.58 MIPS and the average complexity 18.24 MIPS by using (3.1) and (3.2) respectively.

3.3.2 Limitation of This Implementation

Because of the space limitation of the on-board memory, a very short test vector is used in this implementation. But this VAD-CNG algorithm is very data-dependent, i.e., the complexities and average complexities could vary significantly if different test vectors are used. Figure 3.3 is an example of the data-dependency of this algorithm. Another 4-second test vector is used in the implementation and quite different complexities and average complexities are obtained, compared to those obtained from Figure 3.2.

To overcome the data-dependency and obtain more accurate complexities and average complexities, a relatively longer test vector rather than a 4-second test vector should be used, and this test vector should be stored on the host instead of on the board. Therefore,

the algorithm will be implemented in the next section using the host channels (HST module), which are described in Section 2.2.2.4.



Statistics View

STS	Count	Total	Max	Average
vad_SWI	400	48865550 inst	209438 inst	122163.88
cng_SWI	400	2391200 inst	5978 inst	5978.00
VadAnalyze_STS	400	47862952 inst	206930 inst	119657.38
hpfilter_STS	400	11825836 inst	29566 inst	29564.59
autocorr_STS	399	32021752 inst	80380 inst	80255.02
levdurb_STS	62	1085316 inst	18680 inst	17505.10
analysis_STS	5	158850 inst	31770 inst	31770.00

Figure 3.3. Statistic View of the implementation using another on-board test vector

3.4 Implementation Using Host Channels

In this section, the modified algorithm is implemented using the HST and PIP modules provided by the DSP/BIOS. Instead of using the actual device to send and receive analog speech signals, the HST uses pipes internally to receive the input data flow from one host file and send the output data flow to another. The HST method is chosen because of two reasons. (1) It is very convenient to modify the program when we want to use some peripheral devices other than the host PC. When we are ready to modify the program to use the peripheral devices, we can retain the code that manages the target's end of the pipe and rewrite the code of the functions that handle the device I/O to manage the other end of the pipe. (2) Longer test vectors could be used because they are stored on the host instead of on the board. More accurate test results could be obtained from this implementation.

Using the Host Channel Manager of the Configuration Tool, two HST objects **input_HST** and **output_HST** are created and bound to the input file and output file respectively, as shown in Figure 3.4. The input file contains a 6-minute 12-dB speech test

vector (28.8×10^6 words) and is stored on the host to provide the input data for the application. The output file is another host file to store the output data of the application.

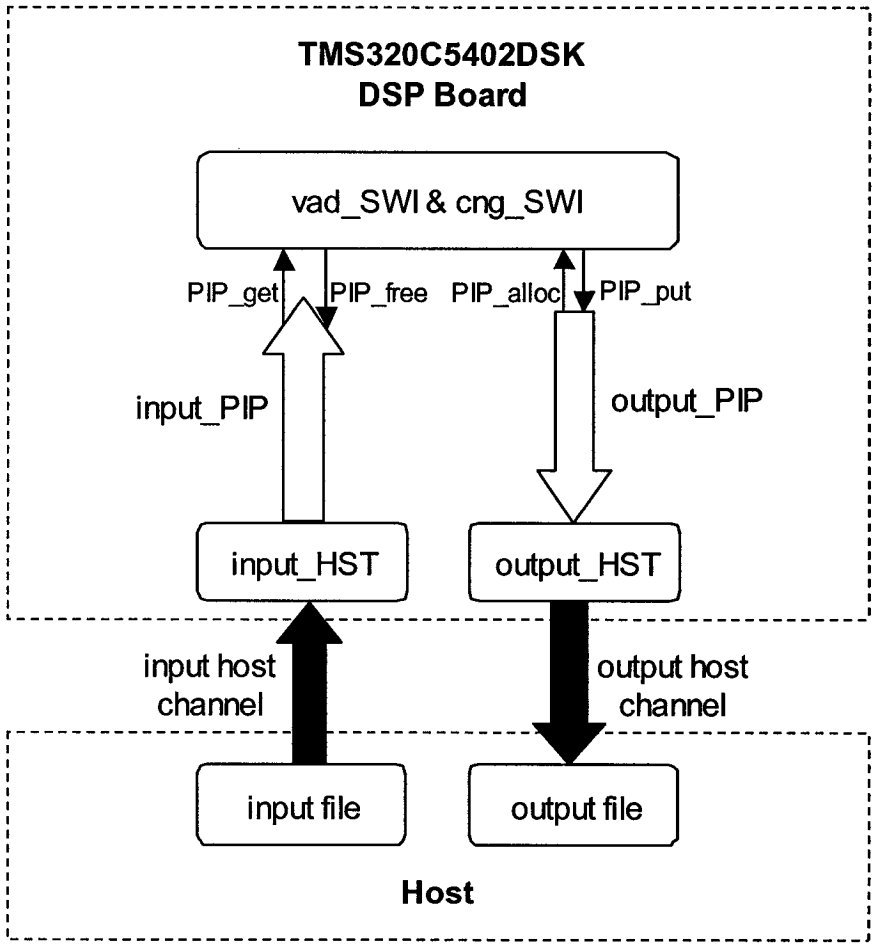


Figure 3.4. Block diagram of implementation using host channels

These two HST objects will trigger the SWI, **vad_SWI**, by clearing the mailbox for **vad_SWI**, when a full frame of data for the input channel and free space of a frame for the output channel become available. The SWI object is posted when the mailbox value becomes zero. The mailbox for **vad_SWI** is set to 3 (0011). The smallest bit of the mailbox is cleared when the input channel contains a full frame and the second smallest bit is cleared when the output channel contains an empty frame. In this way, **vad_SWI** is posted

only when there is a full frame available in the input channel and an empty frame available in the output channel. And **cng_SWI** is triggered each time the program exits **vad_SWI**.

These two HST objects use two pipes, **input_PIP** and **output_PIP** respectively by calling the HST_getpipe assembly macro to get the address of the internal PIP object used by each HST object. These two pipes are created dynamically and internally in the *vad* and *cng* modules respectively.

To synchronize the input and output data transfer, some assembly macros described in Section 2.2.2.4 are used. The call to PIP_get gets a full frame from the input pipe and the call to PIP_alloc gets an empty frame from the output pipe. Then, this full frame of input data is processed with the modified VAD-CNG algorithm and the results written into the empty frame obtained from the output pipe. After the whole frame is processed, PIP_put is called to put the full frame back into the output pipe and PIP_free is called to clear and recycle the input frame so that it can be reused the next time.

The memory arrangement is just the same as that of the implementation using an on-board test vector except that the **.data** section is loaded to the **IDATA** segment instead of the **EDATA** segment because there is no more test vector needed to be stored on board. In this case, the total size of all the program sections is 12.536K words, whereas that of all the data sections is 5.714K words, i.e., 18.250K words memory space is needed in this implementation.

3.5 Performance Analysis

The real-time analysis tool, Statistics View, described in Section 2.3 is used to test the performance of the implementation. A 6-minute 12-dB voice test vector generated from the database of the standard TIA/EIA/IS-727 is used as the input to the algorithm during the test.

To compare the performance, both the original and the modified Nortel algorithms are implemented using the host channels with the same input file. The test results are calculated and shown in Table 3.1.

TABLE 3.1. COMPLEXITIES AND AVERAGE COMPLEXITIES OF THE ORIGINAL AND MODIFIED ALGORITHMS (12-DB SNR INPUT)

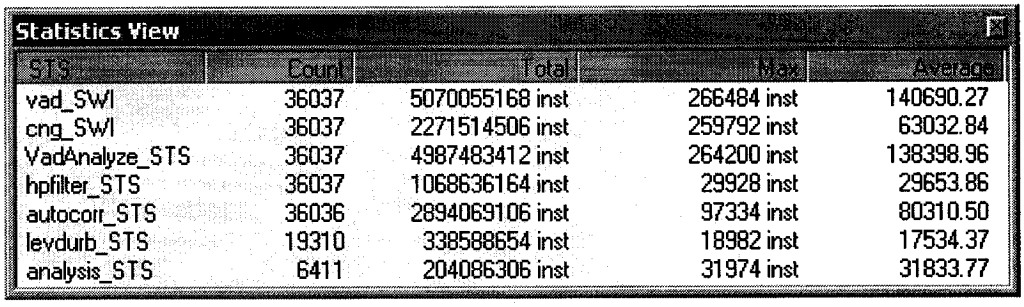
Module/ Algorithm	**Complexity**			**Average Complexity**		
	Original algorithm (MIPS)	Modified algorithm (MIPS)	Reduction (%)	Original algorithm (MIPS)	Modified algorithm (MIPS)	Reduction (%)
vad	40.34	26.65	34	33.33	14.07	58
cng	26.25	25.98	1	6.54	6.30	4
VadAnalyzer	40.11	26.42	34	33.10	13.84	58
VAD_CNG algorithm	66.59	52.63	21	39.87	20.37	49

Table 3.1 shows that for the *vad* module, the complexity has been reduced by 34% from 40.34 MIPS to 26.65 MIPS and the average complexity by 58% from 33.33 MIPS to 14.07 MIPS through the modifications, whereas there is a small change in the complexity or the average complexity of the *cng* module after the modifications. This is because in [4], most of the modifications have been aimed at the *vad* module, or more accurately, its sub-

module *VadAnalyzer*, which has a 34% complexity reduction. For the entire VAD-CNG algorithm, the reduction in complexity is 21%, and that in average complexity is 49% after the modifications. These results are consistent with the test results that have been obtained in [4] from simulations of the original and modified algorithms on a PC.

For the TMS320C5402DSP DSK, which has a 100M Hz CPU and can execute 10^8 instructions per second, theoretically speaking, all the algorithms with complexities less than 100 MIPS could run on it in real-time. In this sense, both the original and the modified algorithms meet the real-time requirement. But, considering that the VAD-CNG algorithm is not designed to be used alone and should be a part of a packet voice communication system, which consists of many integral parts, such as the coder, decoder, echo canceller, and voice enhancement, and all these parts need to work together in real-time, there exists a need to reduce the implementational complexity of the modified algorithm as much as possible through some optimizations.

From the **Total** column in Figure 3.5, it is seen that the *vad* module consumes about $(5.07 \times 10^9) / (5.07 \times 10^9 + 2.27 \times 10^9) = 69\%$ of the computational resources of the entire algorithm. Thus our research will focus on the *vad* module. The *VadAnalysis* module, which is the major part of the *vad* module, accounts for $(4.99 \times 10^9) / (5.07 \times 10^9) = 98\%$ of the *vad* module. From the complexity view, *VadAnalysis* accounts for more than (264200 / 266484) = 99% of the MIPS consumed by *vad*.



Statistics View

STS	Count	Total	Max	Average
vad_SWI	36037	5070055168 inst	266484 inst	140690.27
cng_SWI	36037	2271514506 inst	259792 inst	63032.84
VadAnalyze_STS	36037	4987483412 inst	264200 inst	138398.96
hpfilter_STS	36037	1068636164 inst	29928 inst	29653.86
autocorr_STS	36036	2894069106 inst	97334 inst	80310.50
levdurb_STS	19310	338588654 inst	18982 inst	17534.37
analysis_STS	6411	204086306 inst	31974 inst	31833.77

Figure 3.5. Statistic View of the implementation of the modified algorithm

To find the most computation-consuming modules in *vad*, the MIPS consumed by **vad_SWI** should be broken down further to individual constituent sub-modules of the *vad* module. The real-time analysis tool, Statistics View, is applied to each module described in Section 3.1. From the test results shown in Table 3.2, it is clear that the most MIPS-hogging modules are *autocorr* (9.73 MIPS), *cng_quantize* (5.03 MIPS), *analysis* (3.20 MIPS), *hpfilter* (2.99 MIPS), *cng_update* (2.67 MIPS), and *levdurb* (1.90 MIPS), which have a total complexity of 25.52 MIPS.

Actually, the *cng_quantize* and *cng_update* modules, which are used to implement the CNG encoder functionality, belong to the CNG algorithm even though they are included in the *vad* module. The *cng_update* module calls the *levdurb* module, and *levdurb* is the major part of it. Therefore, if the complexity of *levdurb* is reduced, that of *cng_update* is reduced too. The *cng_quantize* module includes a lot of sub-modules and should be optimized in conjunction with the dequantization module in the *cng* module. Thus, its optimization is left as part of a future work to be carried out. Therefore, the optimizations carried out in the next chapter will target the *autocorr*, *hpfilter*, *levdurb* and *analysis* modules.

TABLE 3.2. COMPLEXITIES AND AVERAGE COMPLEXITIES OF C MODULES OF *VAD*

Module name	Complexity (MIPS)	Average complexity (MIPS)
hpfilter	2.99	2.97
autocorr	9.73	8.03
levdurb	1.90	1.75
lintodb	0.16	0.13
d_lr	0.47	0.44
SoftDecision	0.16	0.13
peakTrack	0.04	0.01
minTrack	0.06	0.03
analysis	3.20	3.18
v_magsq	0.68	0.65
cng_update	2.67	2.58
cng_quantize	5.03	4.60
sid_pack	0.05	0.03

3.6 Conclusion

The modified algorithm has been implemented directly using an on-board vector as the input data. This test vector loaded on the DSP board should be very short because of the space limitation of the on-board memory. Because of the data-dependency of this VAD-CNG algorithm, the experimental results obtained from this implementation are not very accurate.

In order to obtain more accurate test results, the original and the modified algorithms are implemented using the host channels. Longer test vectors could be used because they

are stored on the host instead of on the board. The test results on the direct implementation show that the modified algorithm has a 21% reduction in the complexity and a 49% reduction in the average complexity compared to those of the original algorithm. These results are consistent with the test results that have been obtained in [4] from the simulations of the original and modified algorithms on a PC [4].

The test results also demonstrate that both the original and modified algorithms can be implemented in real-time on the DSP board. But, considering that the VAD-CNG algorithm is not designed to be used alone and should be a part of a vocoder algorithm, which should work in real-time, there exists a need to reduce the implementational complexity of the modified algorithm as much as possible. Therefore, some measures such as rewriting, in assembly language, some of the routines that consume large computational resources and making use of the assembly-optimized functions in the DSP Library and the intrinsic functions available on the board, should be undertaken to further reduce the implementational complexity. These optimizations should target the most MIPS-hogging modules.

CHAPTER 4

OPTIMIZED IMPLEMENTATION OF THE MODIFIED ALGORITHM

In this chapter some optimizations are introduced in the direct implementation of the modified algorithm to reduce its implementational complexity and then, some tests carried out to evaluate the effects of these optimizations.

4.1 Introduction

As mentioned in Chapter 3, since the modified algorithm is not designed to be used alone and should be a part of a packet voice communication system, which consists of many integral parts, such as the coder, decoder, echo canceller, and voice enhancement, and all these parts need to work together in real-time, there exists a need to reduce the implementational complexity of this algorithm as much as possible. Therefore, some optimizations are carried out to further reduce the implementational complexity.

The following section describes five measures that will be taken to achieve this end. These optimizations target the most computation-consuming modules in the *vad* module, which are *autocorr*, *hpfilter*, *levdurb* and *analysis*. The optimizations are carried out at the implementation level rather than at the algorithm level.

4.2 Optimizations of the Direct Implementation

The five optimizations carried out on the direct implementation are: (1) performing the program-level optimization by using the optimizer provided by the compiler of the Code Composer Studio (CCS), (2) rewriting the major parts of the most computationally intensive sub-module in *vad* in assembly language, (3) using intrinsic functions in one of the routines that consumes the most computational resources, (4) making use of the assembly-optimized functions in the DSP Library (DSPLIB) to replace some of the general-purpose functions in the algorithm, and (5) reorganizing the memory.

4.2.1 Performing Program-level Optimization

The TMS320C54x C/C++ compiler accepts American National Standards Institute (ANSI) standard C as well as C++ source code and produces assembly language source code for the TMS320C54x device. The compiler tools include an optimization program (optimizer) that improves the execution speed and reduces the size of C/C++ programs by performing such tasks as simplifying loops, rearranging statements and expressions, and allocating variables into registers [32].

The way to invoke the optimizer is to use the cl500 shell program specifying the –o n option on the cl500 command line. The n denotes the level of optimization (0, 1, 2, and 3), which controls the type and degree of optimization [32]:

–o0

- Performs control-flow-graph simplification
- Allocates variables to registers

- Performs loop rotation
- Eliminates unused code
- Simplifies expressions and statements
- Expands calls to functions declared inline

–o1

Performs all –o0 optimizations, plus:

- Performs local copy/constant propagation
- Removes unused assignments
- Eliminates local common expressions

–o2

Performs all –o1 optimizations, plus:

- Performs loop optimizations
- Eliminates global common subexpressions
- Eliminates global unused assignments
- Performs loop unrolling

–o3

Performs all –o2 optimizations, plus:

- Removes all functions that are never called
- Simplifies functions with return values that are never used
- Inlines calls to small functions

- Reorders function declarations so that the attributes of called functions are known when the caller is optimized
- Identifies file-level variable characteristics

Program-level optimization is applied to the algorithm by using the –pm option with the –o3 option. With program-level optimization, all the source files are compiled into one intermediate file called a module, which is passed to the optimizer and code generator of the compiler. Since the compiler can see the entire program, it performs several optimizations that are rarely applied during file-level optimization [32]:

- If a particular argument in a function always has the same value, the compiler replaces the argument with that value and passes the value instead of the argument.
- If a return value of a function is never used, the compiler deletes the return code in the function.
- If a function is not called directly or indirectly by the main program, the compiler removes the function.

Program-level optimization can be controlled by using the –op option. Specifically, the –op option indicates if a module's external functions can be called or the module's external variables can be modified in other modules. The number following –op indicates the levels described as follows [32]:

Use this option	**If the module**
–op0	Has functions that are called from other modules and global variables

that are modified in other modules

–op1 Does not have functions that are called by other modules but has global variables that are modified in other modules

–op2 Does not have functions that are called by other modules or global variables that are in other modules

–op3 Has functions that are called from other modules but does not have global variables that are modified in other modules

The –op2 option is used in the algorithm. So the compiler should be set as in Figure 4.1.

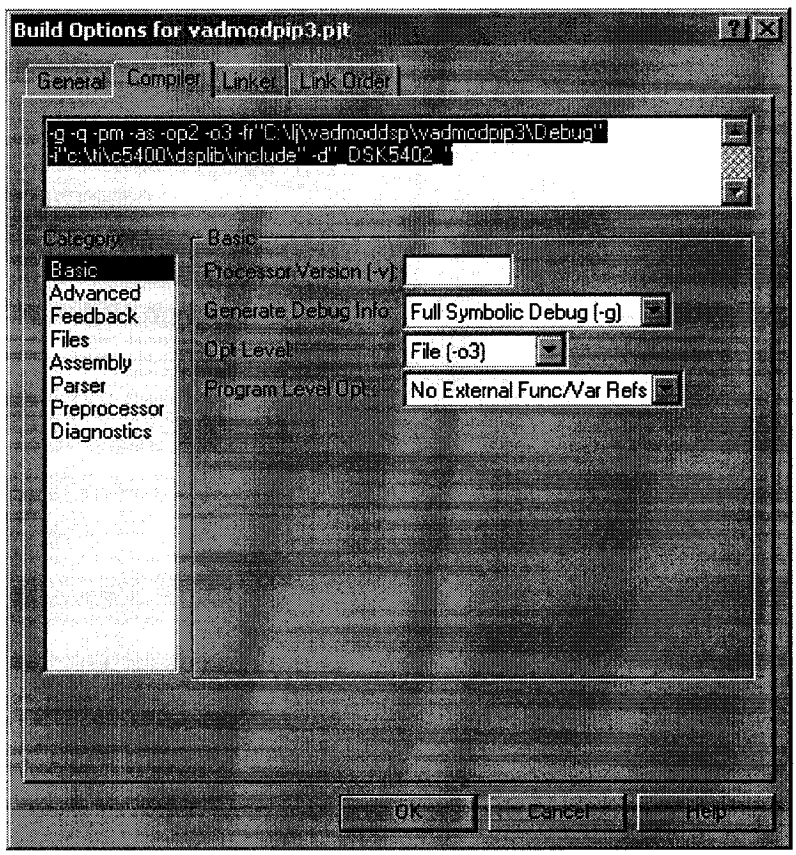


Figure 4.1. Setting the complier

Since the compiler recognizes only the C/C++ source code but not any assembly code that might be present, the –pm –o3 –op2 option will exclude the C/C++ functions called from an assembly function and the variable modifications to C/C++ functions in an assembly function [32]. In order to keep these functions in the program, the "FUNC_EXT_CALLED Pragma" is introduced to specify to the optimizer to retain these C functions or any other functions called by these C functions by adding "#Pragma FUNC_EXT_CALLED (function name);" before any declarations or references to these functions.

This optimization is applied to the entire algorithms. Table 4.1 gives the complexities and the average complexities of the different modules of the modified algorithm. This table shows that the efficiency of the code for the entire algorithm, as well as for each module or sub-module, has improved significantly through this optimization. The implementational complexity of the modified algorithm is reduced by 39% from 52.63 MIPS to 31.93 MIPS, and the average complexity by 45% from 20.37 MIPS to 11.18 MIPS.

4.2.2 Rewriting Some C Routines in Assembly Language

The TMS320C54x C/C++ compiler accepts C or C++ source code and produces assembly language source code. Unfortunately, the efficiency of the assembly code produced by the complier from C/C++ is not comparable to that of the assembly code written directly. Hence, it is valuable to write the most computationally intensive parts of the algorithm in the assembly language, called mixed programming, to get a better program efficiency, even though the assembly programming is not as convenient as programming in the C language.

TABLE 4.1. COMPLEXITIES AND AVERAGE COMPLEXITIES REDUCTION THROUGH PERFORMING PROGRAM-LEVEL OPTIMIZATION (12-DB SNR INPUT)

Module/ Algorithm	Complexity			Average Complexity		
	Before optimization (MIPS)	After optimization (MIPS)	Reduction (%)	Before optimization (MIPS)	After optimization (MIPS)	Reduction (%)
vad	26.65	16.02	40	14.07	7.55	46
cng	25.98	15.91	39	6.30	3.63	42
autocorr	9.73	5.24	46	8.03	3.88	52
levderb	1.90	1.50	21	1.75	1.37	22
hpfilter	2.99	1.96	34	2.97	1.93	35
analysis	3.20	1.37	57	3.18	1.35	58
Modified algorithm	52.63	31.93	39	20.37	11.18	45

From Section 3.5, it is known that the most computation-consuming C module in the *vad* module is *autocorr*, which calculates the autocorrelation coefficients $r(i),(i=0,1,2,\cdots,p)$. The two most computationally intensive parts in this sub-module, the ones that calculate $r(0)$ and $r(i),(i=1,2,\cdots,p)$, are rewritten in the assembly language. The source code can be found in the assembly files autor0.asm and autori.asm in Appendix B.

The computation of the autocorrelation coefficients involves a lot of multiply/accumulate (MAC) operations (see Section 1.2.1.1). In the modified algorithm, $r(0)$ is obtained from the summation of 240 32-bit products, since the window length is chosen as 240. These products are calculated by multiplying 16-bit windowed speech samples (see (1.7)). When these MAC operations are programmed in the C language,

overflow control is necessary since the maximum number of bits that may be required to represent the summation of 240 32-bit products is $(32 + \log_2 240) = 40$ without overflow [35]. The overflow detection is performed from time to time during the accumulation. When an overflow error is detected, the products are recalculated and rescaled (by ½) to get a scaled summation. This kind of overflow can be avoided when programming in the assembly language, since the assembly instructions can control the two independent 40-bit accumulators directly. These accumulators can contain the summation of up to 256 32-bit numbers without overflow. In this way, the overflow detection as well as numerous repeating computations is avoided. Similarly, the rescaling of the products is not needed in the computation of $r(i), (i = 1,2,\cdots, p)$, since the overflow will not happen during the accumulation. The 40-bit summation is normalized to 32-bit by using an exponent encoder that can compute the exponent of a 40-bit accumulator value in a single cycle, and a barrel shifter to execute the shift.

The assembly instruction set can directly control the on-chip devices, such as the multiplier, accumulator, barrel shifter, exponent encoder, and auxiliary register. This is the reason why the assembly language is much more efficient than C, as well as other high-level languages. For example, although multiplication and accumulation are two distinct operations, each normally requiring a separate instruction cycle, the two operations can be executed in a singe cycle if the MAC unit (consisting of a multiplier and an accumulator) can be controlled directly. At a time when the multiplier is computing a product, the accumulator accumulates the product of the previous multiplication. If N products are to be accumulated, N - 1 multiplies can overlap with accumulation. Thus, it takes a total of $N + 1$ instruction execution cycles to compute the sum of products of N multiplications [35]. If N

is large, pipelined MAC operations can be executed at a speed of nearly one MAC operation per instruction cycle. The barrel shifter, used in scaling the variables and parameters, is more efficient than the conventional shift register since it can execute shift by several bits in a single cycle. The exponent encoder can compute the exponent of a 40-bit accumulator value in a single cycle. Furthermore, in assembly programming, the intermediate results can be directly assigned to the on-chip registers rather than the memory to improve the speed of accessing these intermediate results.

Through this optimization, the complexity of the *autocorr* module is reduced by 94% from 5.24 MIPS to 0.30 MIPS, while the average complexity reduced by 93% from 3.88 MIPS to 0.27 MIPS. These test results show that the assembly programming is an efficient way to reduce the implementational complexity of the modified algorithm.

4.2.3 Using Intrinsic Functions

The CCS provides some intrinsic functions that are written in assembly language and can be called by routines coded in C/C++ language [32]. Therefore, it is very convenient to use intrinsics in C program and get a more efficient code at the same time.

The intrinsics are used in *levdurb*, a computationally intensive module. These intrinsics and their corresponding C54x assembly language instructions are listed in Table 4.2. The functions executed by calling these intrinsics are also described.

The experimental results show that through this optimization, the complexity of the *levdurb* module is reduced by 13% from 1.50 MIPS to 1.30 MIPS, while the average complexity has been reduced by 12% from 1.37 MIPS to 1.20 MIPS.

TABLE 4.2. INTRINSIC FUNCTIONS USED IN ALGORITHM

Intrinsic	Assembly Instruction	Description
short _abss(short src);	ABS	Creates a saturated 16-bit absolute value. _abss(0x8000) => 0x7FFF
long _labss(long src);	ABS	Creates a saturated 32-bit absolute value. _labss(0x8000000) => 0x7FFFFFFF
short _rnd(long src);	RND ADD	Rounds src by adding 2^{15}. Produces a 16-bit saturated result.
short _sadd(short src1, short src2);	ADD	Adds two 16-bit integers, producing a saturated 16-bit result.
long _lsadd(long src1, long src2);	ADD	Adds two 32-bit integers, producing a saturated 32-bit result.
long _smac(long src, short op1, short op2);	MAC	Multiplies op1 and op2, shifts the result left by 1, and adds it to src. Produces a saturated 32-bit result.
short _smacr(long src, short op1, short op2);	MACAR	Multiplies op1 and op2, shifts the result left by 1, adds the result to src, and then rounds the result by adding 2^{15}.
long _smas(long src, short op1, short op2);	MAS	Multiplies op1 and op2, shifts the result left by 1, and subtracts it from src. Produces a 32-bit result.
short _smpy(short src1, short src2);	MPYA	Multiplies src1 and src2, and shifts the result left by 1. Produces a saturated 16-bit result.
long _lsmpy(short src1, short src2);	MPY	Multiplies src1 and src2, and shifts the result left by 1. Produces a saturated 32-bit result.
short _smpyr(short src1, short src2);	MPYR	Multiplies src1 and src2, shifts the result left by 1, and rounds by adding $2^{15}$to the result.
short _sshl(short src1, short src2);	SFTA	Shifts src1 left by src2 and produces a 16-bit result. The result is saturated if src2 is less than or equal to 8.
long _lsshl(long src1, short src2);	SFTA	Shifts src1 left by src2 and produces a 32-bit result. The result is saturated if src2 is less than or equal to 8.

4.2.4 Using Assembly-Optimized Functions in DSPLIB

The TMS320C54x™ DSPLIB is an optimized DSP function library for C programmers on TMS320C54x devices. It includes over 50 C-callable assembly-optimized general-purpose

signal-processing routines. These routines are typically used in computationally intensive real-time applications, where optimal execution speed is critical. By using these routines, considerably faster execution speeds can be achieved than by equivalent code written in standard ANSI C language [33].

The *hpfilter* module executes the functionality of a highpass filter, and the *analysis* module executes the functionality of an inverse LPC filter, which is, actually, a finite impulse response (FIR) filter. Both the highpass filter and the FIR filter are widely-used general-purpose DSP routines and are available in DSPLIB. The DSPLIB functions used in the algorithm are *iircas4* (cascaded IIR direct form II using 4-coefficients per biquad) and *fir* (FIR filter), which respectively replace the original routine of the highpass filter and the FIR functions.

4.2.4.1 Cascaded IIR Direct Form II Using 4-Coefficients per Biquad

This function is expressed as [33]

```
short oflag = iircas4(DATA *x, DATA *h, DATA *y, DATA
**dbuffer, ushort nbiq, ushort nx);
```

The conventions and arguments used in this function are described as follows:

DATA	Data type definition equating a short, 16-bit value representing a Q15 format number
x[nx]	Pointer to input data vector of size nx
h[4*nbiq]	Pointer to filter coefficient vector with the following format: **h** = a11 a21 b21 b11 ....a1I a2I b2I b1I

where I is the biquad index (i.e. a21: is the a2 coefficient of biquad 1)
Pole (recursive) coefficients = a
Zero (non-recursive) coefficients = b

y[nx] Pointer to output data vector of size nx.

dbuffer[2*nbiq] Pointer to address of delay line **d**
Each biquad has 2 delay line elements separated by nbiq locations in the following format:
d1(n-1), d2(n-1),.., dI(n-1), d1(n-2), d2(n-2)...dI(n-2)
where I is the biquad index .
This array should be initialized to 0 for the first block only.

nbiq Number of biquads

nx Number of elements of input and output vectors

oflag Overflow flag
· If oflag = 1, a 32-bit overflow has occurred
· If oflag = 0, a 32-bit overflow has not occurred

This function computes a cascade IIR filter of n-biquad (1 biquad in the algorithm) sections. Each biquad section is implemented using direct-form II. All biquad coefficients (4 per biquad) are stored in vector **h**. The real data input is stored in vector **x**. The filter output result is stored in vector **y**. This function retains the address of the delay filter memory **d** containing the previous delayed values to allow a consecutive processing of the blocks.

The algorithm of this function is:

$$d(n) = x(n) - a1 * d(n-1) - a2 * d(n-2) \quad (4.1a)$$

$$y(n) = d(n) + b1 * d(n-1) + b2 * d(n-2) \quad (4.1b)$$

This function is called in the *hpfilter* module to replace the exiting highpass filter routine and it reduces the complexity of *hpfilter* by 89% from 1.96 MIPS to 0.22 MIPS and the average complexity by 90% from 1.93 MIPS to 0.19 MIPS.

4.2.4.2 FIR Filter

This function is expressed as [33]

```
oflag = short fir (DATA *x, DATA *h, DATA *r, DATA
**dbuffer, ushort nh, ushort nx);
```

The conventions and arguments used in this function are described as follows:

DATA	Data type definition equating a short, 16-bit value representing a Q15 format number.
x[nx]	Pointer to real input vector of nx real elements.
h[nh]	Pointer to coefficient vector of size nh in normal order: **h** = b0 b1 b2 b3 …
r[nx]	Pointer to real input vector of nx real elements. In-place computation (**r** = **x**) is allowed.
Dbuffer[nh]	Pointer to address of delay buffer **d** This array should be initialized to 0 for the first block only. Between consecutive blocks, this buffer preserves the previous elements needed.

Nx	Number of real elements in vector **x** (input samples)
Nh	Number of coefficients
Oflag	Overflow error flag · If oflag = 1, a 32-bit overflow has occurred · If oflag = 0, a 32-bit overflow has not occurred

The *fir* function computes a real FIR filter (direct-form) using coefficients stored in vector **h**. The real data input is stored in vector **x**. The filter output result is stored in vector **r**. This function retains the address of the delay filter memory **d** containing the previous delayed values to allow consecutive processing of blocks.

The algorithm of this function is:

$$r[j] = \sum_{k=0}^{nh-1} h[k]x[j-k] \qquad 0 \le j < nx \qquad (4.2)$$

This function is called in the *analysis* module to replace the exiting FIR filter routine and it reduces the complexity of *analysis* by 85% from 1.37 MIPS to 0.21 MIPS and the average complexity by 87%from 1.35 MIPS to 0.18 MIPS.

4.2.5 Reorganizing the Memory

The TMS320C5402DSK includes 16K words (16-bit) of on-chip dual-access RAM (DARAM), 4K words of on-chip ROM and 64K words of external single-access RAM (SARAM). On-chip memory accesses are more efficient than off-chip (external) memory accesses, because there are eight different internal buses for the on-chip memory accesses

on the TMS320C5402, whereas there is only one external bus for the off-chip memory accesses [34]. This means that an off-chip operation requires more machine cycles than that of an on-chip operation. As mentioned in Section 2.1.3, each DARAM block can be accessed twice per machine cycle. Therefore, the CPU and the peripherals can read from and write to a DARAM memory address in the same cycle, while each SARAM or ROM block is accessible once per machine cycle for either a read or a write operation [20]. Thus, it will have a faster execution speed, whenever it is possible, to place the program on DARAM.

A program executed on the TMS320C5402 board is generated by CCS and loaded to the target (TMS320C5402) to run on it. As mentioned in Section 3.4, the generated executable program of the modified algorithm loaded to the TMS320C5402 is so large that it is impossible to place the entire program on DARAM, before optimizations (1) to (4) are done; the **.text** section, which includes the executable code, is placed on the off-chip memory. The memory map for the direct implementation is shown in Figure 4.2(a). **IPROG** and **IDATA** are two 8K-word memory segments defined on the on-chip DARAM, while **EPROG** and **EDATA** are 32K-word segments on the external SARAM. 4K-word memory space of the **EPROG** segment is mapped onto the on-chip ROM.

Through the optimizations (1) to (4), the code size of the generated program is reduced from 18.250K words to 15.505K words. Therefore, it is possible to place the major part of the program, which concerns the complexity of the algorithm, on the DARAM after reorganizing the memory. As shown in Figure 4.2(b), the size of **IDATA** is decreased from 8K words to 4.5K words to give more space to **IPROG** increasing from 8K to 11.5K.

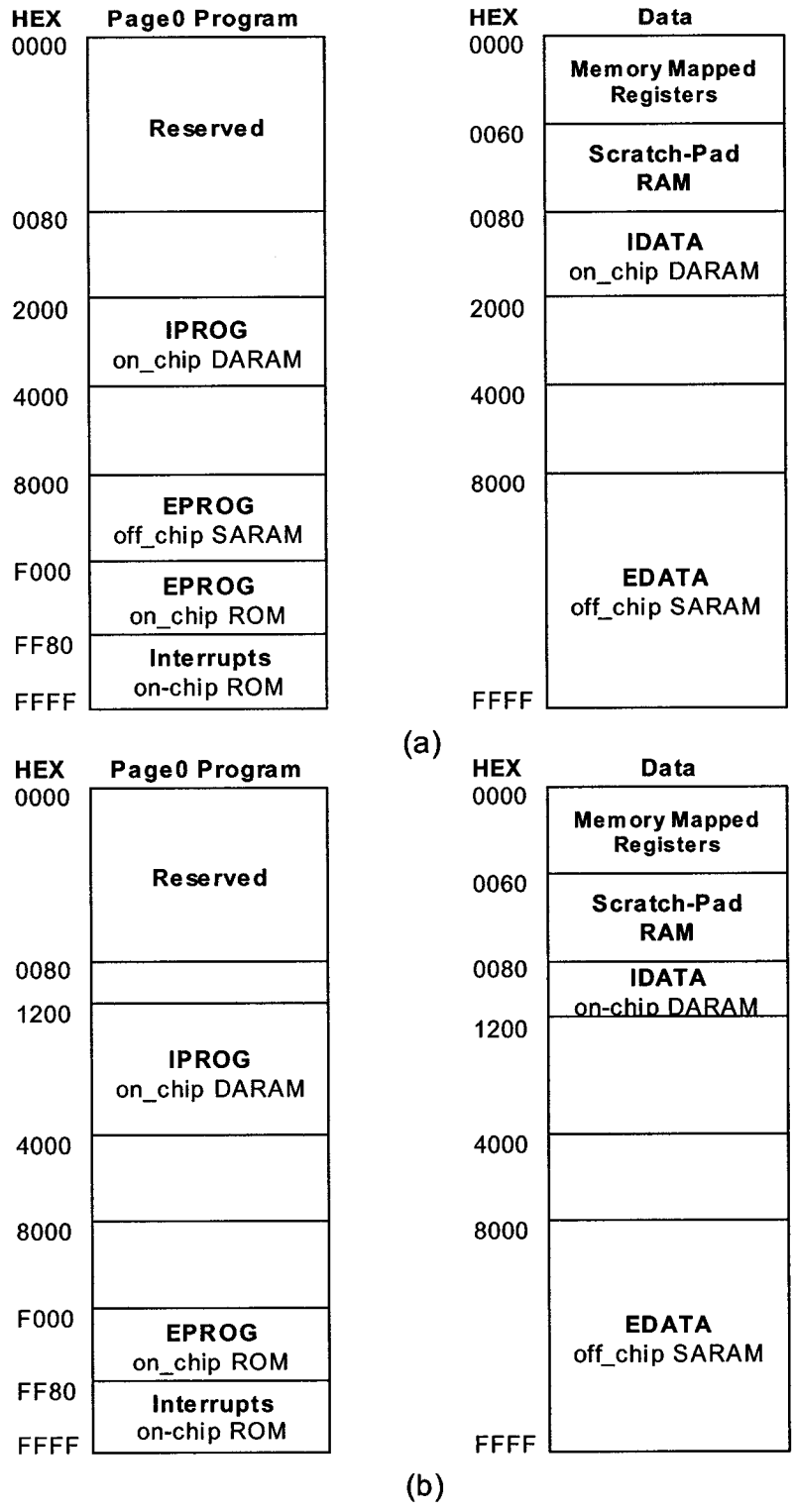


Figure 4.2. Memory maps (a) before reorganizing memory (b) after reorganizing memory

This is done since the 4.5K data segment is big enough to place all the data sections (4.225K words) of the program, whereas the 11.5K **IPROG** memory space is used to hold the 11K program sections. Some parts of the **.text** section, the code for the *ppVadInit*, *ppCngInit* and *ppVadCfg* sub-modules, are executed only once in the program to initialize and configure the program before processing the input data and contribute nothing to the complexity of the algorithm. Thus, these parts (0.28K) are placed in **EPROG** and the rest of the parts of the **.text** section are placed in **IPROG**. The **EPROG** segment is redefined to a 4K space only on the on-chip ROM. After the reorganization, only the on-chip memories are used to place the program and data, and all the sections concerning the complexity are on the DARAM. The speed to read from and write to the memory is much faster, which will lead to an improvement in performance with regard to the speed of the entire algorithm. The reductions in the complexity and the average complexity of the modified algorithm, as well as of each module, are shown in Table 4.3.

TABLE 4.3. COMPLEXITIES AND AVERAGE COMPLEXITIES REDUCTION THROUGH Reorganizing MEMORY (12-DB SNR INPUT)

Module/ Algorithm	**Complexity**			**Average Complexity**		
	Before optimization (MIPS)	After optimization (MIPS)	Reduction (%)	Before optimization (MIPS)	After optimization (MIPS)	Reduction (%)
vad	9.16	3.70	60	1.90	0.81	57
cng	15.91	5.97	62	3.63	1.36	63
autocorr	0.30	0.16	47	0.27	0.13	52
levderb	1.30	0.55	58	1.20	0.49	59
hpfilter	0.22	0.10	55	0.19	0.07	63
analysis	0.21	0.13	38	0.18	0.10	44
Modified algorithm	25.07	9.67	61	5.53	2.17	61

4.3 Experimental Results

To test the efficiency of the optimizations, some voice samples with different SNRs generated from the database of the standard TIA/EIA/IS-727 [29] are used as the inputs to the algorithm before and after the optimizations. The effect of each optimization has already been described in Section 4.2. The composite effect of these optimizations is shown in Table 4.4.

TABLE 4.4. COMPLEXITIES AND AVERAGE COMPLEXITIES REDUCTION THROUGH OPTIMIZATIONS (12-DB SNR INPUT)

Module/ Algorithm	**Complexity**			**Average Complexity**		
	Before optimizations (MIPS)	After optimizations (MIPS)	Reduction (%)	Before optimizations (MIPS)	After optimizations (MIPS)	Reduction (%)
vad	26.65	3.70	86	14.07	0.81	94
cng	25.98	5.97	77	6.30	1.36	78
autocorr	9.73	0.16	98	8.03	0.13	98
levderb	1.90	0.55	71	1.75	0.49	72
hpfilter	2.99	0.10	97	2.97	0.07	98
analysis	3.20	0.13	96	3.18	0.10	97
Modified algorithm	52.63	9.67	82	20.37	2.17	89

From Table 4.4, it can be concluded that the complexity of the modified algorithm is reduced by 82% from 52.63 MIPS to 9.67 MIPS and the average complexity by 89% from 20.37 MIPS to 2.17 MIPS. The complexity of the *vad* module is reduced by 86% and that of the *cng* module by 77%. The *auotcorr* module, whose major parts have been rewritten in the assembly language, has its complexity reduced by 98% from 9.73 MIPS to 0.16 MIPS.

The *levderb* module, which uses the intrinsic functions, has its complexity reduced by 71%. The *hpfilter* module, which calls the *iircas4* function from DSPLIB, has its complexity reduced by 97% from 2.99 MIPS to 0.10 MIPS. Finally, the *analysis* module, which calls the *fir* function from DSPLIB, has its complexity reduced by 96% from 3.20 MIPS to 0.13 MIPS.

The test results in Table 4.4 are obtained using a 12-dB SNR test vector as the input. Tables 4.5, 4.6 and 4.7 show the corresponding complexities and average complexities before and after the optimizations for 6-dB, 18-dB and infinite-dB SNR inputs.

TABLE 4.5. COMPLEXITIES AND AVERAGE COMPLEXITIES REDUCTION THROUGH OPTIMIZATIONS (6-DB SNR INPUT)

Module/ Algorithm	**Complexity**			**Average Complexity**		
	Before optimizations (MIPS)	After optimizations (MIPS)	Reduction (%)	Before optimizations (MIPS)	After optimizations (MIPS)	Reduction (%)
vad	27.04	3.75	86	15.42	1.14	93
cng	26.47	6.01	77	13.24	2.95	78
autocorr	8.13	0.17	98	8.02	0.13	98
levderb	1.92	0.55	71	1.76	0.49	72
hpfilter	2.99	0.10	97	2.97	0.07	98
analysis	3.20	0.13	96	3.18	0.10	97
Modified algorithm	53.51	9.76	82	28.66	4.09	86

TABLE 4.6. COMPLEXITIES AND AVERAGE COMPLEXITIES REDUCTION THROUGH OPTIMIZATIONS (18-DB SNR INPUT)

Module/ Algorithm	Complexity			Average Complexity		
	Before optimizations (MIPS)	After optimizations (MIPS)	Reduction (%)	Before optimizations (MIPS)	After optimizations (MIPS)	Reduction (%)
vad	26.50	3.78	86	15.58	1.18	92
cng	26.45	5.98	77	15.62	3.44	78
autocorr	9.84	0.17	98	8.03	0.13	98
levderb	1.91	0.56	71	1.76	0.49	72
hpfilter	2.99	0.10	97	2.97	0.07	98
analysis	3.20	0.13	96	3.18	0.10	97
Modified algorithm	52.95	9.76	82	31.20	4.62	85

TABLE 4.7. COMPLEXITIES AND AVERAGE COMPLEXITIES REDUCTION THROUGH OPTIMIZATIONS (INFINITE-DB SNR INPUT)

Module/ Algorithm	Complexity			Average Complexity		
	Before optimizations (MIPS)	After optimizations (MIPS)	Reduction (%)	Before optimizations (MIPS)	After optimizations (MIPS)	Reduction (%)
vad	26.26	3.62	86	13.42	0.66	95
cng	25.98	5.92	77	14.51	3.20	78
autocorr	9.64	0.16	98	7.95	0.13	98
levderb	1.95	0.65	67	1.74	0.49	72
hpfilter	3.00	0.10	97	2.97	0.07	98
analysis	3.22	0.13	96	3.18	0.10	97
Modified algorithm	52.24	9.54	82	27.83	3.86	86

From Tables 4.4 to 4.7, it can be concluded that this VAD-CNG algorithm is very data-dependent, i.e., the complexities and average complexities could vary significantly if different test vectors are used, just as discussed in Section 3.3.2. Combining the test results from different SNR test vectors, we conclude that the optimizations reduce the implementational complexity of the algorithm by 82% from 53.51 MIPS to 9.76 MIPS and the average complexity by 86 % from 27.02 MIPS to 3.69 MIPS.

4.4 Summary

In this chapter, five optimizations have been carried out on the direct implementation of the modified algorithm on TMS320C5402DSK DSP board. The experimental results show that significant improvement has been achieved for the performance of the algorithm in terms of the execution speed. As a consequence of these optimizations, the complexity and the average complexity have been reduced by 82% and 86%, respectively. With these optimizations, the modified VAD-CNG algorithm can be incorporated into a practical real-time voice communication system.

Chapter 5

A REAL-TIME DEMONSTRATION SYSTEM

A real-time audio codec system with silence suppression is built in our laboratory. This chapter describes the details of this system in terms of the hardware interfaces and software structures.

In the implementation using the host channels, digital data from the files on the host are processed through the use of SWI and the results stored in the files on the host. In this real-time audio system, the digital data will come from sampled analog signals and the output data converted to analog signals. Therefore, a real-time system with real I/O peripheral devices is built. This system can also be used to demonstrate the improvements achieved through the optimizations proposed in the previous chapter.

5.1 Overview of the Real-time System

The real-time system consists of three parts, an audio source, a DSP board and a speaker, as shown in Figure 5.1. The analog speech signals from the audio source are passed through the A/D converter of the codec on the DSP board to be sampled and digitized to 16-bit, 8000 Hz linear PCM data. These digital signals are processed frame by frame by the real-time application, and the output digital data converted to analog signals through the D/A converter and passed to the speaker, which can be used to provide a subjective evaluation of the performance of the modified VAD-CNG algorithm.

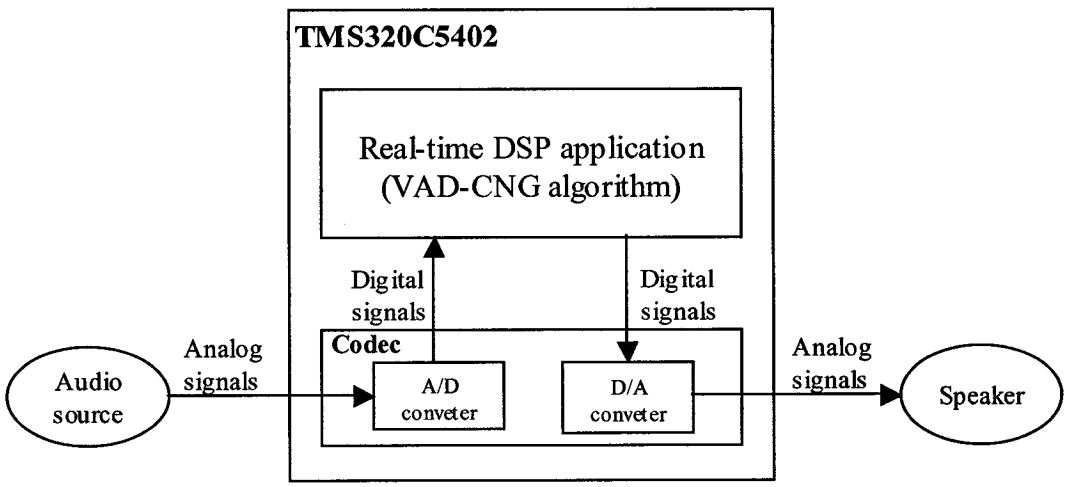


Figure 5.1. Block diagram of the real-time system

5.2 Hardware Interfaces and Dataflow

The hardware connection of this system is shown in Figure 5.2. The digital speech samples, which are stored on the host, are played with a media player, Cool Editor 2000, to be used as the audio source to generate the analog speech signals. This audio source, rather than a microphone, is used because these generated analog signals can be duplicated, i.e., the same analog signals can be obtained repeatedly as long as the same digital speech samples are used and the same volume is set. This is absolutely essential for the tests to be carried out later. The analog input (microphone interface) of the DSP board is connected to the speaker output of the sound card of the PC to receive the analog speech signals generated. The analog output (speaker interface) of the DSP board is connected to the speaker to transmit the output analog signals. The microphone and speaker interfaces (via 3.5 mm audio jacks) of the DSP board are connected to a TLC320AD50 codec that is used to generate 16-bit digital data before the processing and analog signals after the processing.

The TLC320AD50 codec is connected to one of the Multi-channel Buffered Serial Ports (McBSP1) [19].

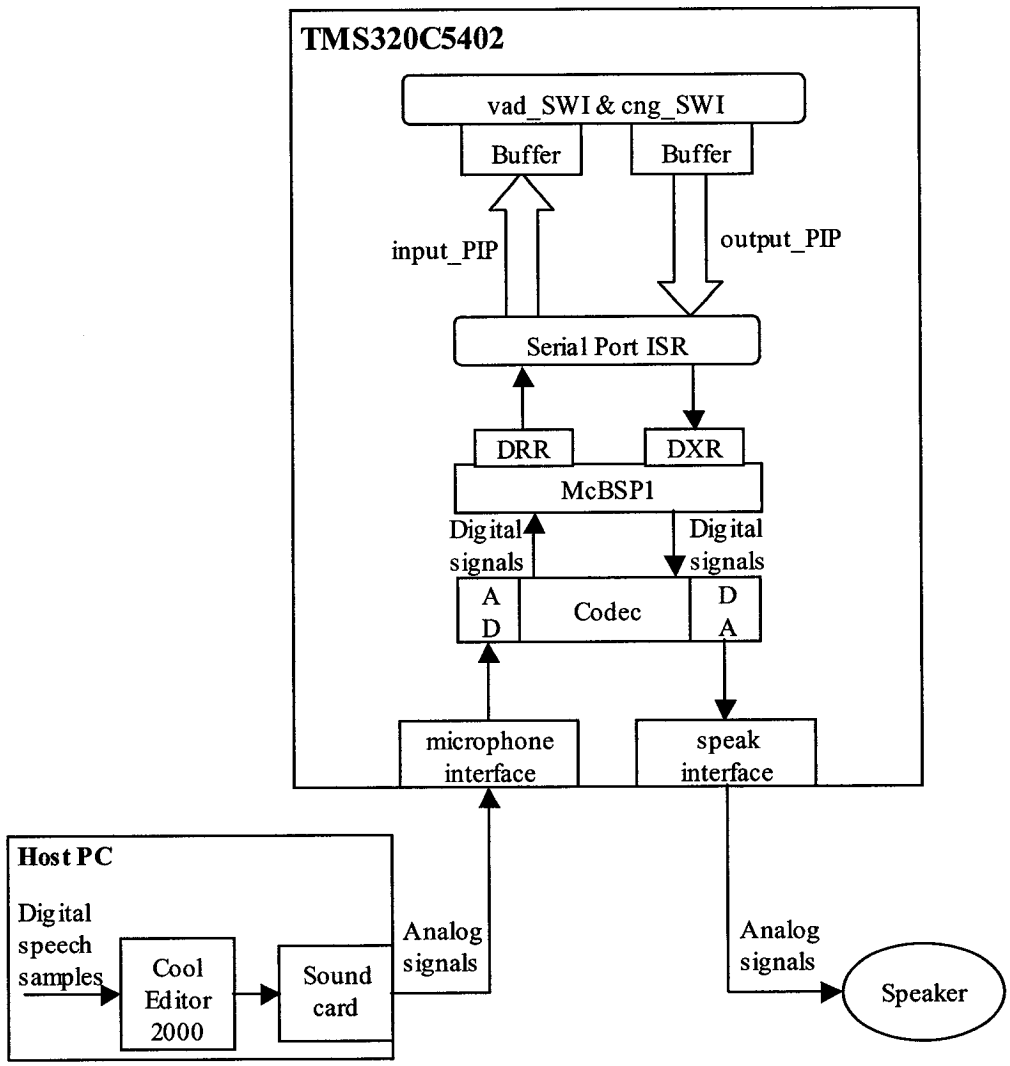


Figure 5.2. Hardware connection and dataflow of the real-time system

As shown in Figure 5.2, the analog signals are sampled and converted to digital signals by the A/D converter on the DSP board through a hardware interrupt (HWI), the serial port interrupt service routine (ISR). The 16-bit, 8000 Hz digital data from the codec flows from the McBSP1 through the input pipe, **input_PIP**, to the application (**vad_SWI** and

cng_SWI), where the digital data are processed frame by frame and sent back to the serial port ISR through the output pipe, **output_PIP**, and then transmitted to the speaker through the codec after being converted to analog signals by the D/A converter.

The serial port ISR copies each new 16-bit data sample in the data receive register (DRR) to a frame of **input_PIP**. When the frame is full (80 samples), the ISR puts the frame back into **input_PIP** that will be read by **vad_SWI**. The function also writes a frame to **output_PIP** each time the **cng_SWI** is triggered and the serial port ISR writes a 16-bit word from this frame to the data transmit register (DXR) each time. The empty frame is recycled back to the **output_PIP** for reuse after the whole frame has been transmitted [30].

5.3 DSP Scheduling

This application is scheduled with a HWI (**HWI_SINT10**) and two SWIs (**vad_SWI** and **cng_SWI**).

5.3.1 HWI in the Application

A hardware interrupt (HWI) **HWI_SINT10** is triggered every 1/8000 second to call the McBSP1 ISR, *DSS_isr*, a standard assembly ISR of McBSP1 provided by TI that can be found in dss_aisr.s54 (see Appendix C), which handles receiving data from and sending data to the codec at the rate of 8000 samples per second. If an **input_PIP** frame is full, the ISR calls the PIP_put assembly macro to put the frame back to the pipe. If an **output_PIP** frame has been transmitted, the ISR will call the PIP_free assembly macro to clear and recycle back this frame to **output_PIP**. The ISR also calls *DSS_rxPrime* to allocate the next empty frame from **input_PIP** after it has done filling up a frame and *DSS_txPrime* to

get the next frame from **output_PIP** if available after it has done transmitting a frame (see Section 5.3.2). The **HWI_SINT10** is the default McBSP1 receive interrupt [35] and is defined as in Figure 5.3.

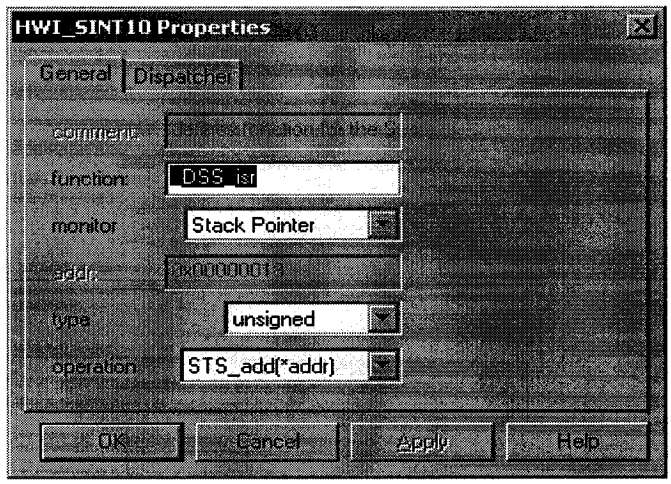


Figure 5.3. Defining HWI_SINT10

5.3.2 SWIs in the Application

There are two SWIs in this application, **vad_SWI** and **cng_SWI**. The **vad_SWI** SWI is triggered through two pipes object, **input_PIP** and **output_PIP**, which are defined as in Figure 5.4 and Figure 5.5, respectively.

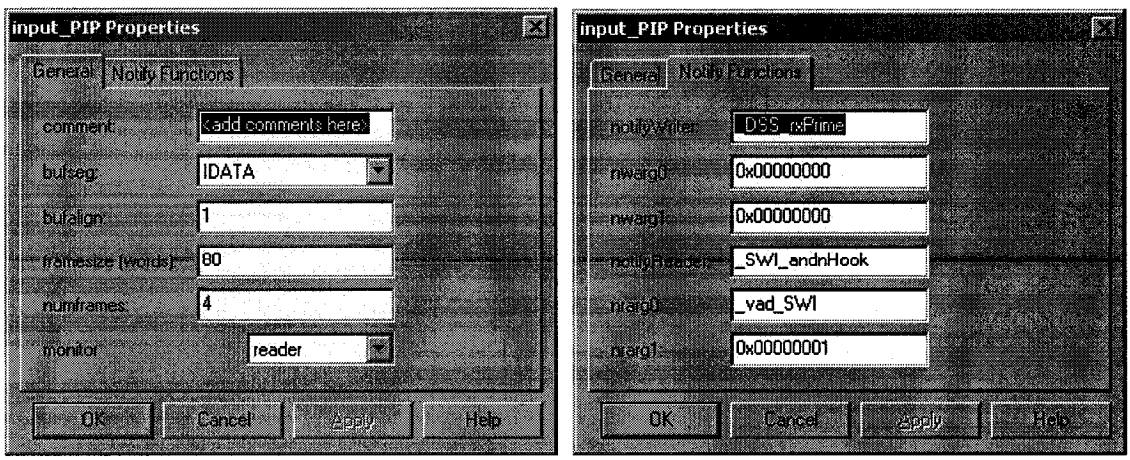


Figure 5.4. Defining input_PIP

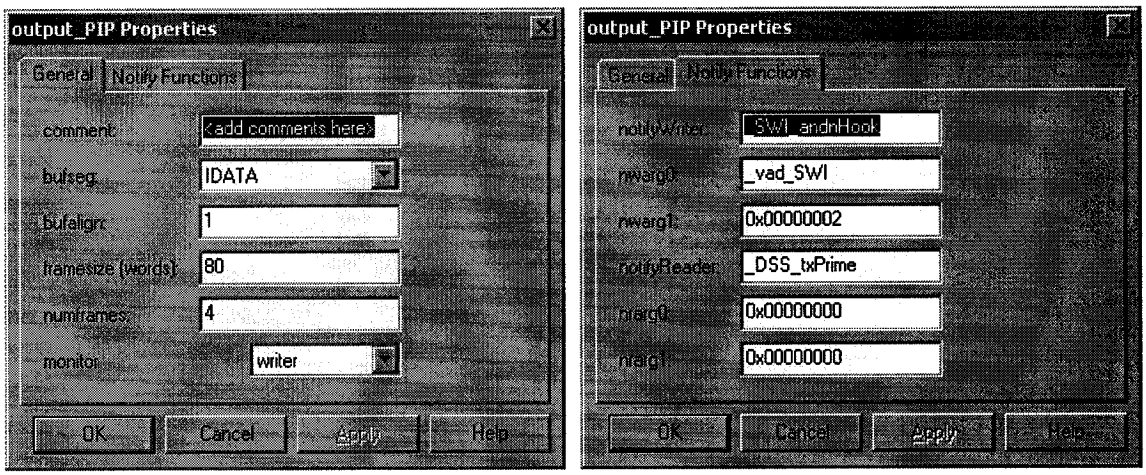


Figure 5.5. Defining output_PIP

Both the input and the output pipes have a 4-frame (80 words per frame) buffer defined in the **IDATA** memory segment. The buffer of the input pipe is used to temporarily store the input data before they are processed, whereas that of the output pipe is used to temporarily store the output data before they are transmitted to McBSP1. The notifyReader function *SWI_andnHook* of the **input_PIP** will clear the first bit in the mailbox for the **vad_SWI** when a full frame is put into the **input_PIP**, whereas the notifyWriter function *SWI_andnHook* of the **output_PIP** will clear the second bit in the mailbox for the **vad_SWI** when an empty frame is available in the **output_PIP**. The mailbox for **vad_SWI** is set to 3 (0011). As mentioned in Chapter 3, the SWI is posted when the mailbox value becomes zero. In this way, **vad_SWI** is posted only when there is a full frame available in **input_PIP** and an empty frame available in **output_PIP** [30]. Since the sampling rate is 8000 Hz and the frame length is 80, **vad_SWI** is triggered every 10 ms to execute the *vad* module. The *vad* module calls PIP_get macro to get a full frame of data from the input pipe and this frame of data are processed to obtain the VAD decision and the SID packet, if necessary. After the processing, PIP_free macro is called to clear and recycle the input

frame so it can be reused in the next time. The **cng_SWI** SWI is posted each time the program exits **vad_SWI**. Thus, it is also triggered every 10 ms. It uses the outputs of the *vad* module as inputs to execute the *cng* module. The *cng* module calls PIP_alloc macro to get an empty frame from the output pipe. It generates the output speech signal by copying the input speech samples, if the VAD output is 1, or by synthesizing the comfort noise, if the VAD output is 0. At the end of **cng_SWI**, PIP_put macro is called to put the full frame back into the output pipe.

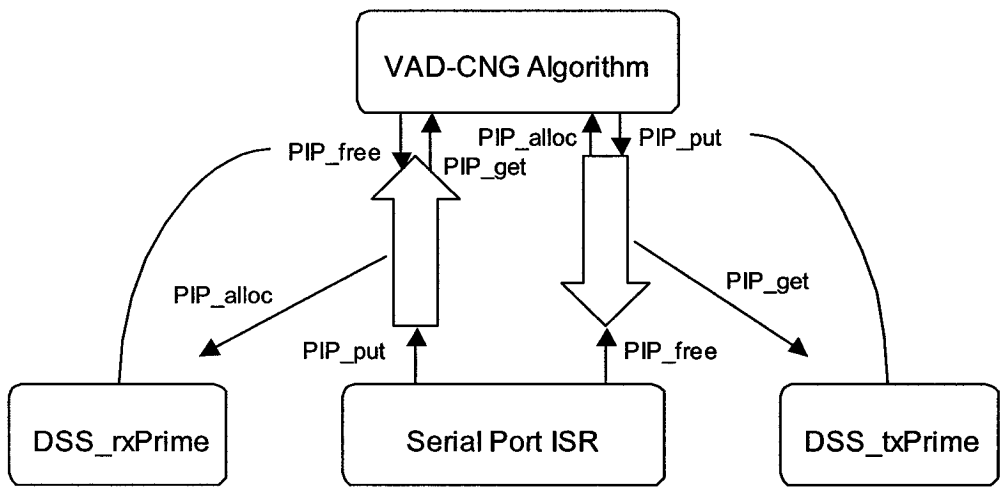


Figure 5.6. DSS_rxPrime and DSS_rxPrime

The notifyWriter function for **input_PIP**, *DSS_rxPrime*, and the notifyReader function for **output_PIP**, *DSS_txPrime*, are C functions that can be found in dss.c (see Appendix C). As shown in Figure 5.6, *DSS_rxPrime* calls PIP_alloc to allocate an empty frame from **input_PIP** that will be used by the ISR to write the data received from the codec. *DSS_rxPrime* is called whenever an empty frame is available in **input_PIP** and the ISR is done with the previous frame. *DSS_txPrime* calls PIP_get to get a full frame from **output_PIP**. The data in this frame will be transmitted by the ISR to the codec.

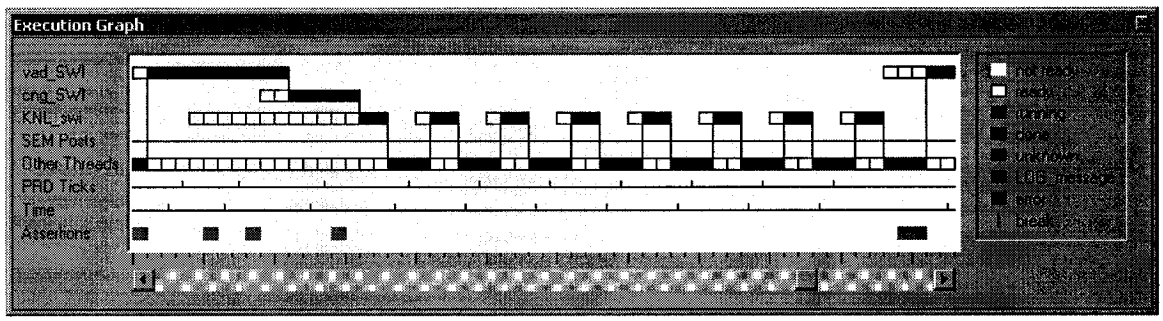
DSS_txPrime is called whenever a full frame is available in **output_PIP** and the ISR finish transmitting the previous frame. Obviously, both *DSS_txPrime* and *DSS_txPrime* will be called every 10 ms.

5.4 Experimental Results

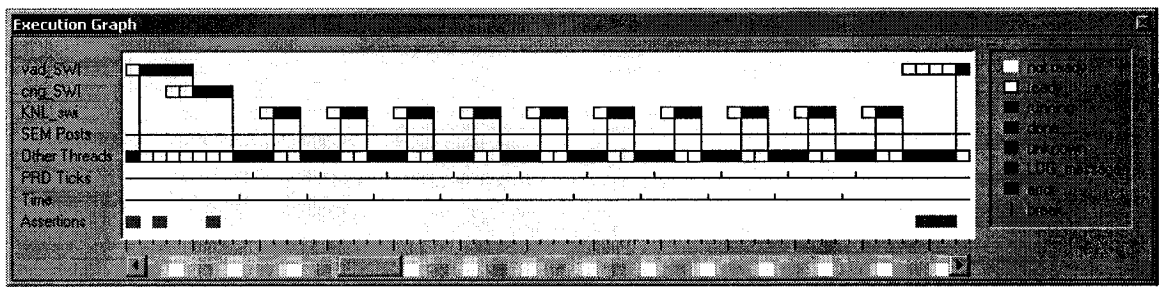
Two real-time analysis tools described in Chapter 2, namely, the Execution Graph and CPU Load Graph, are employed to demonstrate the performance of this real-time system and the improvement achieved through the optimizations proposed in the previous chapter. The real-time system is built based on both the direct and optimized implementations. The same digital speech test vectors are used and the same volume set to make sure that the same analog signals could be obtained, which are the input to the DSP board. In this way, we ensure that the test conditions are the same.

5.4.1 Execution Graph

The Execution Graphs of the real-time system based on the direct and optimized implementations are shown in Figure 5.7, and they are obtained using the same test conditions. By comparing these two graphs, it can be concluded that before the optimizations, the **vad_SWI** and **cng_SWI** are triggered every 10 ms exactly and the average time spent to execute these 2 SWIs is about 2~3 ms, while this runtime is always less than 1ms after the optimizations. This runtime reduction implies that the implementational complexity of the algorithm has been reduced significantly.



(a)



(b)

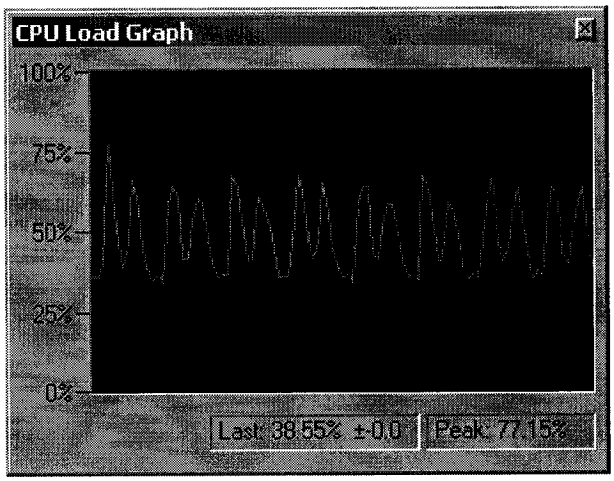
Figure 5.7. Execution Graphs of the real-time system based on (a) direct implementation and (b) optimized implementation

5.4.2 CPU Load Graph

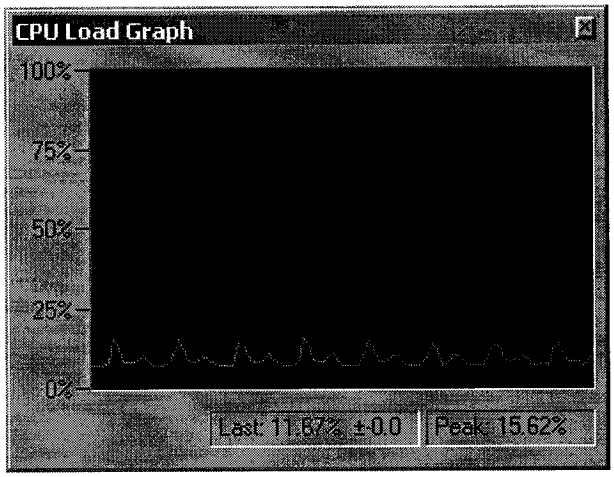
By comparing the CPU Load Graphs before and after the optimizations (see Figure 5.8), it can be concluded that the average CPU load is reduced from about 50% to 10% and the peak value from 86% to 21% through the optimizations. This shows clearly that the CPU load has been reduced significantly through the optimizations.

As mentioned in Chapter 3, this algorithm is very data-dependant. If we change the volume but still use the same digital speech samples, a different CPU load is obtained as shown in Figure 5.9. This is due to the fact that when the volume is changed, the input analog signals to the DSP board are different. Thus, after the sampling and the A/D

conversion, the digital data obtained are also different. With different inputs, different CPU loads are obtained, which implies a change in the execution time, thus illustrating the data-dependency of this algorithm.

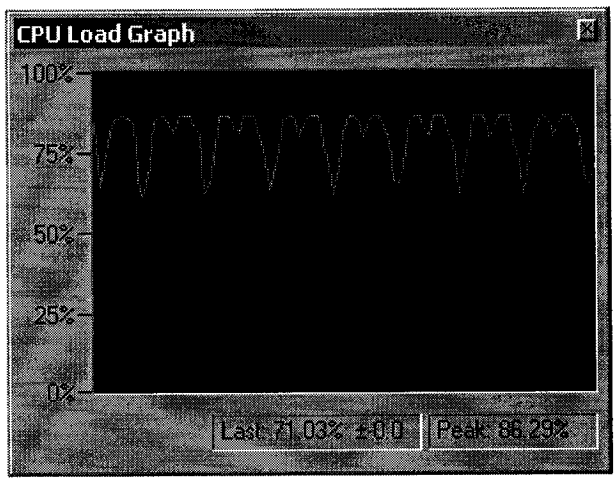


(a)

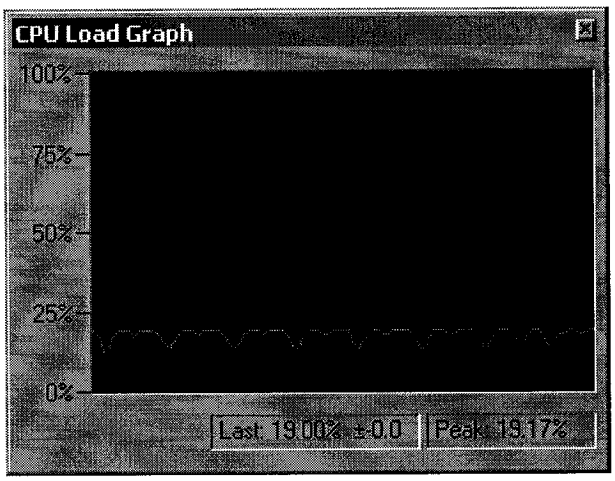


(b)

Figure 5.8. CPU Load Graphs of the real-time system based on (a) direct implementation and (b) optimized implementation



(a)



(b)

Figure 5.9. CPU Load Graphs of the real-time system with different volume based on (a) direct implementation and (b) optimized implementation

Chapter 6

CONCLUSION AND FUTURE RESEARCH DIRECTIONS

6.1 Conclusion

VAD-CNG algorithms are widely employed in packet voice communication systems to reduce transmission bandwidth by suppressing the inactive part of the speech. As an integral part of a vocoder system, the implementational complexity is critical when it is implemented with DSP processors, considering the real-time requirement and the product cost.

In this study, we have focused on a modified version of a Nortel VAD-CNG algorithm to investigate effective ways of implementing it on a DSP board and optimize the implementation in order to reduce the complexity. The modified algorithm has been first directly implemented on a TMS320C5402DSK DSP board using host channels. The test results showed that the modified algorithm could be implemented in real-time on this DSP board. Since a VAD-CNG algorithm is designed to work as a part of a vocoder algorithm, one should aim at further reducing the implementational complexity of the modified algorithm. To this end, five optimizations targeting the most computationally intensive sub-modules in the *vad* module have been carried out to reduce the implementational complexity of the modified algorithm. Experimental results have shown that the complexity has been reduced by 82% from 53.51 MIPS to 9.76 MIPS and the average complexity by 86% from 27.02 MIPS to 3.69 MIPS, where the complexity (average complexity) is the maximum (average) number of million instructions executed per second

(MIPS). In order to demonstrate as to how this algorithm is implemented in real-time with real I/O peripheral devices, a real-time system has been built in the laboratory. The experimental results from the real-time system verify the effectiveness of the optimizations.

6.2 Future Research Directions

Further research can be conducted towards reducing the implementational complexity reduction of the modified Nortel VAD-CNG algorithm for it to be incorporated into a vocoder.

It has been our goal to reduce the implementational complexity of the modified algorithm without or with as little a degradation in its performance. We have investigated several measures to achieve this reduction. Using the intrinsic functions provided by DSP boards is a convenient, but not very effective means of reducing the complexity. Using the assembly-optimized functions in the DSP Library is both convenient and effective, but it can be only applied to general–purpose functions with appropriate data format. Assembly language programming, though not very convenient, can be a very effective method of reducing the complexity and applied to the entire modified algorithm.

The MIPS consumed by the *vad* module, as well as by the *cng* module, can be broken down to individual constituent modules. The optimizations that have been carried out in this thesis have been targeted to the *vad* module in reducing the implementational complexity of the modified VAD-CNG algorithm. The *cng* module could also be treated in a similar manner and given greater attention in a future work.

APPENDIX

This appendix includes the source code of the directly implemented modified algorithm, the optimized implemented algorithm and the real-time implemented algorithm (all recorded on a CD), and the description of the source code.

Appendix A. Source Code of Direct Implementation

The source code of the direct implementation of the modified algorithm is zipped into mod_alg.zip, which could be found on the attached CD. The files included in this zip file are listed as follow.

ANSI C files:

vad.c	VAD/CNG interface simulation
ppVInit.c	VAD initialization
ppVCfg.c	VAD configuration
ppVAnal.c	VAD operation
analysis.c	Inverse LPC filtering
autocorr.c	Autocorrelation function
basic_op.c	Basic operations
cng.c	CNG state initialization & parameter update
cng_quan.c	SID information quantization
d_lr.c	Spectral non-stationarity measure
iir_df2.c	highpass filtering

lar_quan.c	LAR quantization
levdurb.c	Levinson-Durbin recursion
lintodb.c	Conversion of linear value to dB
mintrack.c	Minimum energy tracking
peaktrack.c	Peak energy tracking
rctolar.c	Conversion of reflection coefficients to LARs
sid_pack.c	LARs packing
v_bwexp.c	Bandwidth expansion
v_copy.c	Array copying
v_magsq.c	Vector energy calculation
v_set.c	Setting an array to a single value
ppCInit.c	CNG initialization
ppCCfg.c	CNG configuration
ppCGen.c	CNG operation
cng_gen.c	Comfort noise generation
lar_deq.c	LAR inverse quantization
lartorc.c	Conversion of LARs to reflection coefficients
lintoexp.c	Exponential function
rctoacng.c	Conversion of reflection coefficients to predictor parameters
sid_unpk.c	LARs unpacking
squarert.c	Square root function
synth.c	LPC synthesis filtering
v_gauss.c	Gaussian random number generation

v_scale.c	Vector scaling

ANSI head files:

basic_op.h	Prototypes
cng.h	Prototypes
filter.h	Prototypes
ppvad.h	Structures and prototypes
typedef.h	Data type definitions
udef.h	Definitions
vad.h	Structure
vad_lpc.h	Prototypes
vadcngll.h	Low level common structure
cng_gen.h	Prototypes
cng_lpc.h	Prototypes
cng_defs.h	Constants
cngsynth.h	Structure
ppcng.h	Prototypes
vaddefs.h	Constants

Implementation files:

vadmodprofile.pjt	Project file
vadmodprofile.cdb	Configuration of the project
vadmodprofilecfg.cmd	Linker command file generated by the Configuration Tool

vadmodprofilecfg.h	Generated header file included by vad.c, containing declarations of objects created with the Configuration Tool.
vadmodprofilecfg_c.c	Generated C file containing program code for CSL settings.
vadmodprofilecfg.s54	Assembly source code generated by the Configuration Tool
vadmodprofilecfg.h54	Header file generated by the Configuration Tool, which is included by vadmodprofilecfg.s54
vadmodprofile.out	Executable program (fully compiled, assembled, and linked), which can be loaded and run on the target

Appendix B. Source Code of Optimized Implementation

The source code of the optimized implementation of the modified algorithm is zipped into op_alg.zip, which could be found on the attached CD. The files included in this zip file are the same as the source code of direct implementation of the modified algorithm except that some modifications are made in ppVAnal.c, autocorr.c, levdurb.c, analysis.c, iir_df2.c and vadmodprofile.cdb for the optimizations. Two assembly files are added, which are

autor0.asm	Calculation of $r(0)$
autori.asm	Calculation of $r(i), (i = 1,2,\cdots, p)$

Appendix C. Source Code of Real-time Implementation

The source code of the real-time implemented algorithm is zipped into rt_alg.zip, which could be found on the attached CD. The files included in this zip file are the same as the source code of the optimized implementation except that the following files are added to control the sampling, A/D and D/A conversion, and codec.

dss.c	Receiving data from and sending data to the serial port
dss.h	Prototypes and structure
dss_priv.h	Internal implementation declarations
dss_aisr.s54	McBSP1 ISR
dss.h54	Header file included by dss_aisr.s54
dss_dsk5402.s54	Serial port setting
dsscfg.h	Configuration of ISR

REFERENCES

[1] M. N. S. Swamy, M. O. Ahmad, and R. Khatibi, "Low bit rate speech coding", internal report, Concordia University, Montreal, QC, Canada, February 2000.

[2] *Coding of Speech at 8 Kbit/s Using Conjugate-Structure Algebraic-Code-Excited Linear-Prediction (CS-ACELP)*, ITU-T Recommendation G.729.

[3] *Reduced Complexity 8 Kbit/s CS –ACELP Speech Coder*, ITU-T Recommendation G.729, Annex A.

[4] Wei Chu, "A Modified Silence Suppression Algorithm and Its Performance Test", M.A.Sc. thesis, Department of Electrical and Computer Engineering, Concordia University, Montreal, QC, Canada, March 2003.

[5] W. P. Leblanc and S. A. Mahmoud, "Report on voice activity detection for packet voice transport", Carleton University., Ottawa, ON, Canada, Dec. 1997.

[6] S. Zhang, "Development and implementation of the new VAD-CNG" (with corresponding C-code), Nortel Networks, Ottawa, ON, Canada, May 2000.

[7] *A Silence Compression Scheme for G.729 Optimization for Terminals Conforming to Recommendation V.70*, ITU-T Recommendation G.729, Annex B.

[8] R. Tucker, "Voice activity detection using a periodicity measure", in *IEEE Proceedings-I*, Vol.139, No.4, August 1992.

[9] D. K. Freeman, G. Cosier, C. B. Southcott, and I. Boyd, "The voice activity detector for the Pan-European digital cellular mobile telephone service", in Proceedings of *ICASSP 1989*, pp. 369-372.

[10] D. Lee, H. P. Stern, and S. A. Mahmoud, "A voice activity detection algorithm for communication systems with dynamically varying background acoustic noise", in *VTC'98*, pp. 1214-1218.

[11] J. Sohn and W. Sung, "A voice activity detector employing soft decision based noise spectrum adaptation", in Proceedings of *ICASSP 1998*.

[12] J. Sohn, N. S. Kim, and W. Sung, "A statistical model-based voice activity detection", *IEEE Processing Letters*, Vol.6, No.1, Jan. 1999.

[13] E. Nemer, R. Goubran, and S. Mahmoud, "Robust voice activity detection using higher-order statistics in the LPC residual domain", *IEEE Trans. on Speech and Audio Processing*, Vol.9, No.3, pp.217-231, March 2001.

[14] J. D. Markel and A. H. Gray, Jr., *Linear Prediction of Speech*, Springer -Verlag Berlin Heidelberg, 1976.

[15] S. Haykin, *Adaptive Filter Theory*, 3 rd edition, Prentice Hall, 1996.

[16] L. R. Rabiner and R. W. Schafer, *Digital Processing of Speech Signals*, Prentice - Hall, 1978.

[17] D. O'Shaughnessy, *Speech Communications—Human and Machine*, 2nd edition, IEEE Press, 2000.

[18] S. Zhang, "An overview of the new VAD/CNG", Nortel Networks, Ottawa, ON, Canada, May 2000.

[19] *TMS320C54x Code Composer Studio Help*, Texas Instruments, Dallas, TX.

[20] *TMS320C54x DSP Reference Set, Volume 1: CPU and Peripherals*, Texas Instruments, Dallas, TX, March 2001.

[21] *TMS320C54x DSP Reference Set, Volume 2: Mnemonic Instruction Set*, Texas Instruments, Dallas, TX, March 2001.

[22] *TMS320C54x DSP Reference Set, Volume 3: Algebraic Instruction Set*, Texas Instruments, Dallas, TX, March 2001.

[23] *TMS320C54x DSP Reference Set, Volume 4: Applications Guide*, Texas Instruments, Dallas, TX, October 1996.

[24] *TMS320C54x DSP Reference Set, Volume 5: Enhanced Peripherals*, Texas Instruments, Dallas, TX, June 1999.

[25] *Code Composer Studio Getting Started Guide*, Texas Instruments, Dallas, TX, November 2001.

[26] *TMS320 DSP/BIOS User' s Guide*, Texas Instruments, Dallas, TX, November 2001.

[27] *TMS320C5000 DSP/BIOS Application Programming Interface (API) Reference Guide*, Texas Instruments, Dallas, TX, November 2001.

[28] *TMS320C54x Code Composer Studio Tutorial*, Texas Instruments, Dallas, TX, February 2000.

[29] *TDMA Cellular/PCS -Radio Interface-Minimum Performance Standards for Discontinuous Transmission Operation of Mobile Stations*, TIA/EIA/IS-727, June 1998.

[30] Shawn Dirksen, "An Audio Example Using DSP/BIOS", Texas Instruments, Dallas, TX, November 1999.

[31] Lim Hong Swee, "Implementation of G.729 on the TMS320c54x", Texas Instruments Singapore (Pte) Ltd, Singapore, March 2000.

[32] *TMS320C54x Optimizing C/C++ Compiler User's Guide*, Texas Instruments, Dallas, TX, June 2001.

[33] *Optimized DSP Library for C Programmers on the TMS320C54x*, Texas Instruments, Dallas, TX, October 2000.

[34] *TMS320C54x DSP Programmer's Guide*, Texas Instruments, Dallas, TX, July 2001.

[35] Avtar Singh and S. Srimivasam, *Digital Signal Processing — Implementations Using DSP Microprocessors with Examples from TMS320C54xx*, Brooks/Cole — Thomson Learning, 2004.